Libros de Cátedra

Programación E1201

Curso de Grado

Pablo A. García, Marcelo A. Haberman
y Federico N. Guerrero (coordinadores)

FACULTAD DE
INGENIERÍA

e
exactas





PROGRAMACIÓN E1201

CURSO DE GRADO

Pablo A. García
Marcelo A. Haberman
Federico N. Guerrero

(coordinadores)

Facultad de Ingeniería





Este libro va dedicado a “Grace”.
Por su generosidad,
por darnos siempre un lugar,
por habernos facilitado el camino y
por haber conformado la base de este gran grupo humano
que hoy conforma la cátedra.

Agradecimientos

A nuestra querida Facultad de Ingeniería y todo su personal, que nos ha formado como profesionales y hoy en día nos da la posibilidad de devolver parte de lo recibido y formar, a su vez, a profesionales de las distintas ramas de la ingeniería por medio de la educación pública.

También queremos agradecer a las personas que conforman y conformaron la cátedra por su labor y dedicación, colegas que han contribuido a nuestra formación durante nuestros años como profesionales, y estudiantes que han cursado la materia y aportado a la mejora continua de la misma. Una gran parte de este libro está moldeada por sus preguntas y silencios durante las clases y momentos de mayor y menor éxito en los parciales.

Índice

Introducción 7

Prólogo 8
Graciela Toccaceli

Capítulo 1
Organización de la Computadora y representación interna de datos.. 9
Federico N. Guerrero

Capítulo 2
Fundamentos básicos de programación y algoritmos 46
Marcelo A. Haberman

Capítulo 3
Elementos básicos del lenguaje de programación “C” 81
Pablo A. García

Capítulo 4
Tipos de datos básicos y arreglos.. 109
Leandro Mendez, Alejandro Moyano, Juan M. Rosso

Capítulo 5
Funciones 132
Marcelo A. Haberman

Capítulo 6
Punteros 158
Federico N. Guerrero

Capítulo 7
Estructuras ... 175
Pablo A. García

Capítulo 8
Manejo de archivos ... 187
Marcelo A, Haberman

Capítulo 9
Memoria dinámica ... 205
Federico N. Guerrero

Capítulo 10
Introducción a la Programación Orientada a Objetos ... 233
Pablo A. García

Bibliografía ampliatoria ... 265

Los Autores ... 267

Introducción

Este libro surgió con el propósito de contar un libro de referencia para los alumnos de la cátedra Programación-E1201 de la Facultad de Ingeniería de la Universidad Nacional de La Plata. Aún así, el mismo es accesible al público general que desee aprender la disciplina de la programación de computadoras, comenzando por las bases de la programación, la creación de algoritmos con lenguajes sencillos como diagramas gráficos, pasando por las bases del C y abarcando hasta los temas más complejos como la administración de la memoria de la computadora y la programación orientada a objetos.

La relevancia de la propuesta se fundamenta en la importancia actual de la programación y la conveniencia de la proliferación de material con una fuerte orientación didáctica en este tema, que presenta un gran escalón para quienes se inician en él. Si bien existe una rica base bibliográfica, la experiencia de los integrantes de la cátedra adquirida en años de dictado del curso, ha resultado en una perspectiva única que no se refleja en ninguna alternativa disponible.

El principal motivo, es el acompañamiento que se hace de los alumnos desde el desconocimiento absoluto del mundo de la programación hasta técnicas muy avanzadas, mientras que los textos iniciáticos se enfocan muy superficialmente en los temas avanzados, y los textos avanzados no dedican una didáctica suficiente a los primeros pasos que deben darse para comenzar a razonar algorítmicamente.

En segundo lugar, la larga trayectoria de los docentes de la cátedra en la enseñanza de la materia ha resultado en una recopilación de estrategias didácticas depuradas, que han dado muy buenos resultados para la enseñanza de temas complejos para los alumnos como los algoritmos de búsqueda, funciones recursivas, punteros, estructuras auto referenciadas, entre otros.

Por estos motivos hemos decidido plasmar dichas lecciones en un formato disponible para los alumnos por fuera de las horas de clase, y por primera vez para el público general.

Finalmente, la formación de los docentes como Ing. Electrónicos aporta una visión del mundo de la programación distinta, con un fuerte fundamento en el hardware sobre el cual se programa, otorgando un perfil único a la obra propuesta.

Prólogo

Los autores de este libro, queridos amigos y compañeros de años de docencia en la Facultad de Ingeniería de la UNLP, me han honrado al pedirme que escriba este prólogo por lo cual les estoy muy agradecida.

Producto de la experiencia docente por ellos adquirida surge este texto que plasma, a través de explicaciones teóricas acompañadas de ejercicios especialmente seleccionados, los fundamentos necesarios para alcanzar una buena metodología de programación.

El lector encontrará los conocimientos necesarios para lograr un buen diseño de software, utilizando conceptos de ***Programación Estructurada*** combinados con el uso de distintos ***Tipos y Estructuras de Datos***. Estas herramientas le permitirán abordar el diseño de algoritmos de ordenación y búsqueda, métodos numéricos y manejos de caracteres, entre otros. Asimismo, encontrará como optimizar el uso de la capacidad de memoria del ordenador mediante ***Estructuras Dinámicas*** (pilas, colas, listas), empleadas particularmente en aplicaciones para Comunicaciones y Control.

Todos estos conocimientos, brindan el marco adecuado para incursionar en la moderna técnica de ***Programación Orientada a Objetos***, en la cual datos y funciones (objetos) interactúan eficientemente al realizar el procesamiento requerido.

Siendo que los paradigmas de la ***Programación Estructurada*** y la ***Programación Orientada a Objetos*** emplean diversos lenguajes para su codificación, el lector encontrará en el libro una adecuada explicación de los dos lenguajes más utilizados en este contexto, como son el ***C y el C++***.

El lenguaje ***"C"*** es particularmente utilizado en ámbitos universitarios y profesionales para el desarrollo de aplicaciones que emplean ***Programación Estructurada***. Resulta ser simple, elegante, eficiente y muy utilizado en el desarrollo de sistemas operativos, compiladores, sistemas embebidos y en la escritura de casi todos los lenguajes de alto nivel más populares en la actualidad.

El lenguaje ***"C++"*** contiene todas las características y ventajas del ***"C"*** y además permite una eficiente implementación de aplicaciones que hacen uso de la ***Programación Orientada a Objetos***, tales como navegadores web, aplicaciones gráficas y bases de datos, entre otros.

Finalmente resulta importante destacar que este libro constituye un excelente material didáctico, al desarrollar en forma clara y precisa los contenidos necesarios para lograr un buen nivel de programación. Esto permitirá al lector resolver problemas complejos emergentes del ejercicio de la profesión, mediante el uso de un ***Ordenador***.

Prof. Ing. Graciela Toccaceli
Fac. Ingeniería UNLP
La Plata, 14 de octubre de 2020

CAPÍTULO 1
Organización de la Computadora y representación interna de datos

Federico N. Guerrero

Introducción

Este libro trata sobre la programación, tarea que se realiza fundamentalmente en y para una computadora. Por lo tanto, ella será nuestro objeto de estudio en este capítulo: analizaremos a la computadora a grandes rasgos para introducir el tema, luego los conceptos fundamentales para entender su funcionamiento y finalmente estudiaremos en detalle cómo la información del mundo real (números, textos, etc) se almacena en el mundo interno de la máquina.

En ingeniería se suelen usar los conceptos de *top-down* y *bottom-up*, dos formas diferentes de encarar el estudio de un sistema o la implementación de una solución. *Bottom-up* es el método al que estamos más acostumbrados en el sistema académico universitario: se parte de los fundamentos, de las cuestiones más básicas y de mayor detalle y una vez dominadas se progresa a partir de comprender cómo esas partes conforman sistemas más complejos.

La visión *top-down* parte de ver al sistema completo "desde arriba", como vemos el mundo desde un avión: ciudades, campos, industrias y rutas que los unen; desde un "alto nivel de abstracción", vemos a los sistemas divididos en módulos de los que sabemos qué hacen y cómo interactúan entre sí, pero no cómo funcionan o de qué partes están compuestos.

Puede leerse este libro comenzando por este capítulo para comprender estos fundamentos y luego continuar construyendo el conocimiento de programación a partir de estos bloques (estrategia *bottom-up*), o puede sumergirse directamente en el mundo de los algoritmos a partir del capítulo 2 y luego retornar a este primer capítulo para comprender qué sustenta el funcionamiento de lo que se programa (estrategia *top-down*).

A continuación, utilizaremos un enfoque mixto para comprender la computadora. Primero la analizaremos como un grupo de "cajas negras" y luego profundizaremos en los conceptos más básicos de su constitución. En la segunda parte del capítulo, estudiaremos con un enfoque *bottom-up* sistemas de representación: debemos primero re-aprender nuestro sistema de numeración decimal que damos por sentado para luego estudiar el sistema utilizado por la computadora y finalmente cómo utilizar este simple sistema para representar formas complejas de información.

En el texto utilizaremos los términos en inglés *hardware* y *software*, adoptados en el habla común en español para diferenciar el conjunto de elementos físicos que componen la computadora del conjunto de programas que, como veremos, en todo momento se ejecutan en la misma.

La Computadora

Visión Top-Down: Partes que la componen

Una computadora es una "Máquina electrónica que, mediante determinados programas, permite almacenar y tratar información, y resolver problemas de diversa índole"[1]. Esta definición de diccionario es muy acertada para encarar el aprendizaje de la computadora desde el punto de vista de una persona que estudia Ingeniería y aprende a programar.

Nuestros programas se ejecutarán en la computadora y permitirán a un usuario ingresar información a través de ciertos periféricos de entrada, y obtener el resultado del procesamiento de la información a través de periféricos de salida.

La palabra "información" aquí se usa para significar cualquier dato ya sea extremadamente simple como el estado de un botón ("presionado" o "no presionado"), algo intermedio como una página de texto, o elementos complejos y extensos como 7 temporadas de una serie televisiva en alta resolución o la base de datos de todos los ítems en venta en una plataforma de *e-commerce*. Quizás sea conveniente revertir la definición y entender "información", al menos la de tipo digital, como todo lo que puede entrar y salir de una computadora. Esta percepción de la computadora le da nombre a la disciplina de la "informática": procesamiento de **infor**mación en forma auto**mática**.

Muchos dispositivos cumplen la función de una computadora aunque estrictamente no lo son (reciben otros nombres técnicos), pero si se considera un criterio más blando, podemos incluir como computadora a un celular, tablet, e incluso al humilde horno de microondas[2].

Para comenzar a analizar y explicar la computadora, podemos ver las cosas desde el más *alto nivel*: se compone de periféricos, objetos que se conectan con cables a "la computadora", que es una especie de caja negra. El periférico de salida al que estamos más acostumbrados es la pantalla, donde se muestra información de manera visual. Otros muy comunes son los parlantes y los motores que producen vibraciones, constituyendo una salida táctil. Los periféricos de entrada usuales son teclado y *mouse*, pantallas táctiles, mandos de consolas de videojuegos. Para analizar la "caja negra" de la computadora es necesaria una visión de más *bajo nivel* y considerar cuáles son sus componentes internos.

1 Diccionario de la Lengua española, Ed. 2018, Real Academia Española.
2 Los sistemas de cómputo integrados en dispositivos que no son estrictamente computadoras se denominan "sistemas embebidos". Un horno microondas no es una computadora, pero incorpora una pequeña unidad de cómputo electrónica para funcionar.

Los detalles “finos” de un sistema de computación pueden ser demasiados para comprenderlos o utilizarlos en su totalidad, y es conveniente verlos agrupados en *niveles*. El *nivel alto* es la visión más externa del sistema, sin detalles de cómo están implementados los niveles más bajos. El *bajo nivel* está más cerca de la implementación, de los detalles internos. Este concepto sirve para el *hardware* y también para el *software*.

Si se desciende un nivel desde la vista más simple de la computadora, sus partes componentes principales son una *unidad central de procesamiento (CPU)* que tiene la capacidad de buscar y ejecutar un conjunto de instrucciones y una *memoria principal* que contiene las instrucciones que la CPU debe ejecutar junto con los datos sobre los que operará. En su corazón, todos los sistemas de cómputo tienen un *procesador.* En forma muy simplificada, un procesador es un circuito electrónico al que pueden darse en forma secuencial algunas instrucciones simples una después de la otra, junto con datos, para que ejecute las instrucciones y opere sobre los datos.

Una calculadora es una buena analogía de un procesador, ya que tiene un conjunto de instrucciones (representado por las teclas +, -, etc.) y funciona a base de aplicar estas “instrucciones” sobre un conjunto de datos. El conjunto de instrucciones de que dispone el procesador se conoce como “set de instrucciones”, y si bien individualmente las instrucciones son sencillas, al combinarlas en gran cantidad y a gran velocidad se logran los resultados complejos que observamos a diario al utilizar las computadoras. El procesador ejecuta sus instrucciones a un ritmo constante dado por un *reloj del sistema* que emite una cantidad de pulsos por segundo (medida en veces por segundo o Hertzios) para sincronizar todas las acciones de los demás circuitos.

El segundo elemento de importancia en una plataforma de cómputo es la “memoria”, que es el dispositivo donde se almacenan las instrucciones y los datos que se irán suministrando al procesador. Podemos establecer una analogía con una persona visitando el supermercado. La persona no tiene buena memoria y lleva una lista escrita con todo lo que debe comprar. Cada tanto, mira la lista y recuerda dos o tres elementos que junta de las góndolas. En este ejemplo la persona es un procesador que puede ejecutar la instrucción de “buscar un elemento de la góndola y ponerlo en el carrito”. El cerebro es su memoria principal: necesita tener algunos productos en mente para planificar su recorrido y juntarlos en el corto plazo. Si bien esta memoria es volátil (los elementos serán olvidados pronto) y no muy grande, es la memoria que le dice realmente qué hacer en cada momento, sin la cual no podría hacer nada, y por eso se denomina *principal*. La lista del supermercado cumple el rol de una memoria *secundaria*: tiene la ventaja de ser *permanente*, es decir, no se borra al apagar el dispositivo (¿al irse a dormir?) y puede ser muy grande, pero debe ser transferida de a poco a la memoria principal para poder operar.

Normalmente las computadoras que conocemos tienen una memoria principal conocida como “memoria RAM”[3] y una memoria secundaria compuesta por uno o más discos rígidos[4]. En los teléfonos celulares la disposición es similar pero la memoria secundaria es un chip soldado junto al resto de los componentes en lugar de un disco, y muchas veces puede expandirse utilizando una memoria *flash* en formato de tarjeta SD.

En la figura 1.1 pueden observarse una computadora, un celular y un horno de microondas con sus componentes señalados en forma esquemática. La computadora de escritorio tiene componentes de tamaño relativamente grandes; el procesador es un único circuito integrado (IC). El celular concentra varios componentes integrados en un IC especial llamado SoC (*System on Chip*). El microondas, en contraste, tiene un único IC llamado microcontrolador con todo el sistema de computación integrado.

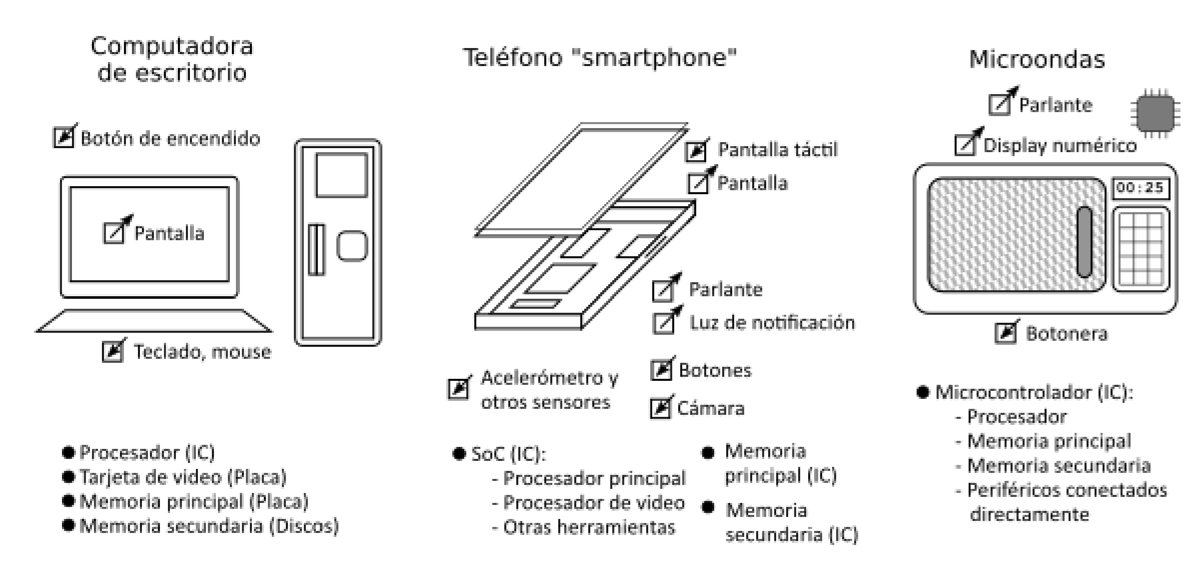


Figura 1.1. *Vista de una PC, un celular, un microondas señalando periféricos de entrada, de salida, y algunos de sus componentes principales.*

Hasta aquí hemos hablado de los componentes físicos o *hardware* de la computadora. Todo esto tiene sentido en tanto se utilice para ejecutar programas, es decir, como soporte para el *software*. Utilizamos los periféricos de la computadora para interactuar con los programas que están corriendo en ella, los cuales nos permiten navegar por la web, ver películas, escribir documentos de texto. Muchos otros programas se ejecutan sin que veamos una interfaz gráfica y se encargan automáticamente de tareas necesarias para que la computadora funcione. Todos estos programas son “fabricados” por *programadores* que escriben el *código fuente* en distintos *lenguajes de programación*.

[3] RAM viene del inglés *Random Access Memory* o “Memoria de acceso aleatorio”, que significa que puede accederse a cualquier dato de cualquier posición en el momento que se quiera, en contraste con otros tipos de memorias en las cuales debe accederse secuencialmente a los datos en algún orden particular.

[4] El nombre “disco” proviene del medio de almacenamiento predominante en las últimas décadas consistente en discos recubiertos de un material magnético. Estas unidades están siendo reemplazadas cada vez más rápidamente por “discos de estado sólido”, que reciben ese nombre a pesar de no contener ningún componente que sea realmente un disco.

Al igual que existen distintas herramientas que pueden usarse para realizar una misma tarea, aún si no corresponden (¿quién no usó alguna vez el mango de un destornillador como martillo?), y de la misma manera que algunas tareas particulares requieren herramientas especializadas, así existen múltiples lenguajes que pueden usarse para el mismo objetivo o que fueron pensados para fines particulares y la elección puede estar fundamentada en razones técnicas o simplemente en gustos personales.

La figura 1.2 muestra ejemplos de segmentos de código fuente escritos en distintos lenguajes.

```c
#include <stdio.h>
#include <stdlib.h>

#define L 80 /* largo*/
#define A 40 /* alto*/
void imprimirBienvenida(void);
void dibujarMundo(char*);

int main()
{
    char mundo[A][L];
    int paleta;
    char input=' ';
    int gmver = 0;
    imprimirBienvenida();
    while(input!='q' && gmver == 0){
        dibujarMundo(&mundo[0][0]);
```

```java
/* Clase para manejar buffers
*  tipo anillo
*  Nombre del archivo:
*  RingBuffer.java
*/
public class RingBuffer{
    public Object[] elements = null;

    private int capacity  = 0;
    private int writePos  = 0;
    private int available = 0;

    public RingBuffer(int capacity) {
        this.capacity = capacity;
        this.elements =
                new Object[capacity];
    }
```

```python
import spidev
import time
import RPi.GPIO as GPIO
import argparse
from pythonosc import osc_message
from pythonosc import udp_client

### Handler de la interrupción de
# Data Ready.

muestra = 0;

def muestras_handler (channel):
    global muestra
    resp = spi.xfer2([0x00, 0x00, 0x00])
    #muestra = 0
    muestra = int.from_bytes(
        [resp[0], resp[1], resp[2]],
        byteorder='big', signed=True)
```

Figura 1.2. *Fragmentos de código fuente escritos en los lenguajes de programación C, Java, y Python.*

A la izquierda de la figura 1.2 se observa un fragmento de código en C. Este lenguaje se popularizó fuertemente en los años 70 e influyó a muchos de los que le siguieron. Sigue siendo muy importante en programación a bajo nivel y su "hijo", C++, es la base de una gran cantidad de software actual de alta eficiencia como juegos y navegadores web. En el medio de la figura se ve Java, el lenguaje quizás más utilizado actualmente por su amplia difusión en dispositivos Android como celulares y *tablets*. Es un lenguaje de una familia conocida como "orientada a objetos". Aprender un lenguaje orientado a objetos es lo que sigue luego de aprender C para cualquier persona que necesite herramientas de programación en ingeniería. A la derecha finalmente se ve Python, otro lenguaje orientado a objetos muy popular en la actualidad.

Cada lenguaje tiene una serie de reglas particulares (*sintaxis*) y palabras clave que es necesario saber para utilizarlo; lo más común es escribir código con ellos, algunos en cambio se utilizan mediante herramientas gráficas. Ese código luego será convertido en un programa capaz de correr en la computadora al ser ingresado a su vez en una segunda línea de programas: compiladores, intérpretes, máquinas virtuales, etc, dependiendo del lenguaje. Para saber más sobre lo que significa esto y cómo se produce la ejecución de los programas veremos en las secciones siguientes el funcionamiento de los sistemas de cómputo desde el bajo nivel.

Visión Bottom-Up: Desde la electrónica al programa

La computadora es un sistema complejo compuesto por la interconexión de circuitos electrónicos digitales[5]. Los circuitos digitales a su vez se basan en bloques relativamente simples creados a partir de *transistores*. En un circuito digital, un transistor cumple la función de una llave que puede controlarse para lograr uno de dos estados y así almacenar o transmitir información. Dos estados como verdadero o falso, $5\ V$ o $0\ V$, alto o bajo, parecen poco, pero combinando muchas de estas llaves puede almacenarse cualquier tipo de información, como lo demuestran los textos, música, videos, incluso impresiones 3D que disfrutamos a diario gracias a nuestras computadoras. La figura 1.3 nos introduce a este concepto de manipular información a través de elementos con 2 estados. Allí se representan números usando llaves que están “para arriba” o “para abajo” y un circuito basado en transistores realiza la suma de estos dos números y los muestra con luces que pueden estar “prendidas” o “apagadas” (se estudiará en detalle esta forma de representar números con dígitos de 2 valores o binarios más adelante en el libro).

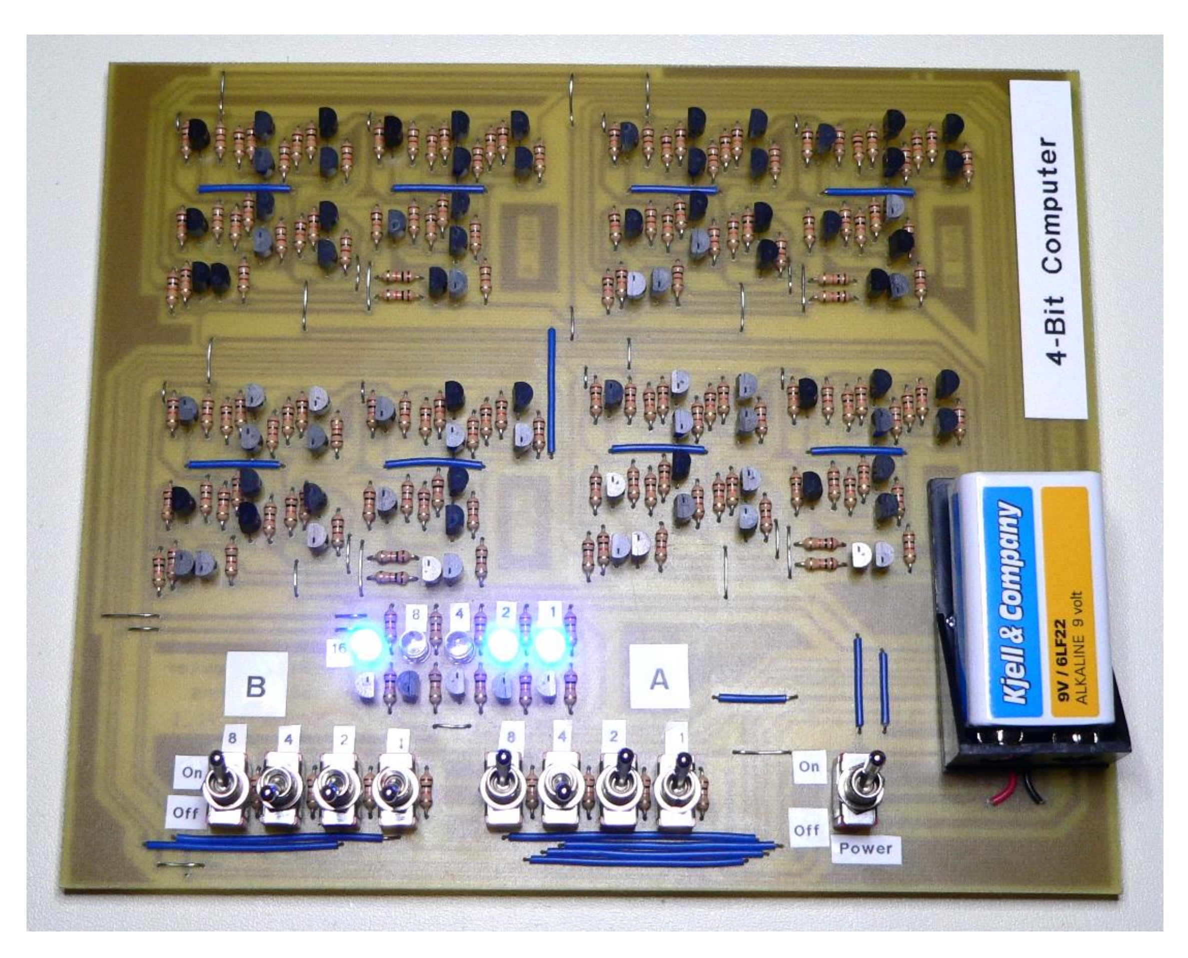


Figura 1.3. *El corazón de los sistemas digitales: circuitos basados en transistores*[6]

[5] En la ingeniería electrónica suele marcarse una división entre los circuitos analógicos, que representan señales con tensiones y corrientes que varían en forma continua (e.g. un amplificador de audio), y los circuitos digitales, que representan las señales con sólo dos niveles de tensiones o corrientes.

[6] Simon Inn. “4-bit computer”, 30 de enero de 2010. Accedido el 13 de abril de 2020. [Blog]. Disponible en https://www.waitingforfriday.com/?p=529. Imagen reproducida con permiso del autor.

La memoria

La unidad mínima de los circuitos digitales es entonces cada una de estas llaves creadas a partir de un conjunto de transistores que permiten almacenar uno de dos valores que representamos con 0 o 1.

Los circuitos digitales de la computadora funcionan con bloques que sólo pueden representar dos niveles: alto y bajo. Cada uno de estos bloques que tiene dos estados posibles se conoce como *bit* (de **B**inary dig**IT**).

Este dígito binario o bit se agrupa en conjuntos conocidos como *registros* (análogamente a como las letras se agrupan en palabras, aunque los registros siempre son del mismo largo). Los registros son circuitos electrónicos que tienen *memoria*, es decir, que pueden retener información.

Una *memoria* es un conjunto de registros donde se almacena información. Se basa en un circuito electrónico capaz de mantener la información en registros y acceder a ella a través de *direcciones*.

La memoria se organiza, en principio, como una gran tabla de celdas cada una con un código que la identifica para que el procesador pueda ubicar cada dato individualmente. Estos códigos o direcciones de memoria no son más que una numeración como la numeración de las casas a lo largo de una calle. En el caso de la memoria principal de la computadora, cada celda puede leerse y escribirse individualmente y con alta velocidad. Algunas memorias principales especiales son sólo de lectura (memorias ROM o *read-only memory*) y suelen usarse para darle una rutina de arranque al procesador. Otras memorias, según la tecnología, pueden ser de mayor tamaño pero sólo leerse de a bloques o muy lentamente, por lo que no sirven para almacenar instrucciones del procesador pero sí grandes volúmenes de datos.

El tamaño de una memoria se mide en función de cuántos bits contiene y es costumbre utilizar multiplicadores que son potencias de 2. Así, la siguiente unidad más grande que el bit es un conjunto de $2^3 = 8$ bits llamada Byte.

Un byte es un conjunto de 8 bits. El bit de la izquierda, el de mayor peso, se conoce como el bit más significativo, y el de la derecha, con menor peso, como el bit menos significativo. Un byte se divide en dos *nibbles* (grupos de 4 bits).

Para hablar de agrupamientos de grandes cantidades de bits se utilizan multiplicadores análogamente al sistema métrico, pero siguiendo con la lógica de potencias de 2. Un agrupamiento de $2^{10} = 1024$bits es 1 kilobit que se escribe como 1 kb, y un grupo de 1024 Bytes es un kiloByte o 1 kB (debe notarse la diferencia entre la b minúscula y mayúscula). Si volvemos a multiplicar por 1024 se tiene 1 megabit (Mb) = 1024 kb, luego 1 Gigabit (Gb) = 1024 Mb y 1 Terabit (Tb) = 1024 Gb. Se suelen utilizar todas estas mismas unidades referidas al Byte en lugar de al bit, y se

escriben kB, MB, GB, TB. En ocasiones las publicidades ofrecen servicios engañosamente refiriéndose a cantidades como "10 Megas", y pensamos que son MB cuando en realidad son Mb. También puede encontrarse en discusiones técnicas que estos multiplicadores se aproximen a miles, millones, etc, pero en realidad corresponden a los mencionados múltiplos de 1024.

El procesador

La computadora almacena información en su memoria, pero debe también poder realizar operaciones sobre esta información, como por ejemplo sumar dos valores numéricos o trasladar un valor de una posición de la memoria a otra. Para eso, se necesitan circuitos especiales que electrónicamente realicen esas operaciones a partir de la manipulación de tensiones y corrientes. Sería imposible implementar estos circuitos para cada registro de la memoria; por lo tanto, se concentra la capacidad de realizar operaciones en un único componente: el procesador, y se le da la habilidad de leer y escribir registros de la memoria, y trasladar esa información a sus registros internos para modificarlos.

El procesador puede realizar ciertas acciones predefinidas sobre sus registros llamadas *instrucciones* que se le otorgan gracias a los circuitos que tenga implementados. Volviendo a la figura 1.3, el circuito mostrado realiza automáticamente la suma de los valores indicados por las llaves; es como un procesador con una única instrucción. En un procesador real se agregan diversos circuitos para que el procesador pueda realizar cada vez más acciones, y se implementan de manera que, al cargar un determinado valor en un registro especial, como si fuese una llave que activa las trabas del tambor de una cerradura, se dispara alguno de los circuitos que implementan las operaciones indicadas por esa instrucción. Se habla de *repertorio o set de instrucciones* para referirse al conjunto de todas las instrucciones disponibles.

De esta manera las instrucciones mismas se almacenan como valores numéricos en la memoria; así, puede almacenarse una lista cuidadosamente elaborada de instrucciones en la memoria (un *programa*) y al encender la computadora, el procesador automáticamente busca estos valores secuencialmente uno por uno (en un proceso que se llama *fetch*). Al recoger cada instrucción, la misma produce en los circuitos del procesador la ejecución de las operaciones (mediante el proceso de *decodificación y ejecución*).

Un ejemplo ilustrativo

Un ejemplo extremadamente simplificado puede ayudarnos a comprender el funcionamiento del procesador y la memoria principal. Imaginaremos un procesador mínimo. Nuestro procesador tendrá sólo un reloj de sistema y dos registros. El primer registro será un "contador de programa", al que llamaremos PC, que indica qué dirección de memoria leer a continuación. El segundo registro será uno de trabajo o "acumulador", que llamaremos W, donde el procesador puede almacenar información temporalmente para realizar operaciones (este registro especial suele estar dentro de una Unidad Aritmético-Lógica o ALU que se encarga de efectuar las operaciones). El funcionamiento básico de este procesador se basa en repetir una y otra vez los siguientes pasos cada vez que el reloj de sistema emite un nuevo pulso:

1. Buscar la instrucción en la dirección de memoria indicada en PC
2. Decodificar y Ejecutar la instrucción
3. Aumentar PC a la siguiente posición de memoria

Ahora que ya tenemos el diseño del *hardware*, dotemos a nuestro procesador del siguiente set de 3 instrucciones, donde `DIR` representa cualquier dirección de memoria:

1. `CARGAR DIR`: Busca el dato que hay en la dirección de memoria `DIR` y lo coloca en el registro `W`.
2. `SUMAR DIR:` Busca el dato que hay en la dirección de memoria `DIR` y lo suma al que está en el registro `W`.
3. `ALMACENAR DIR:` Guarda el contenido del registro `W` en `DIR`.

Recordemos que el procesador es un circuito electrónico que puede realizar estas operaciones porque su hardware está preparado para eso: es como una calculadora mecánica. Pero, como podemos darle una lista de estas instrucciones variándola a nuestro placer, y podemos asignar los valores que queramos a la memoria (a través también de instrucciones que estamos omitiendo por simplicidad aquí) podemos crear distintas secuencias de instrucciones para solucionar diversos problemas (es decir, crear algoritmos) a partir de este reducido set. En la figura 1.4 se ve un ejemplo: realicemos la suma 25 + 5 en nuestra computadora.

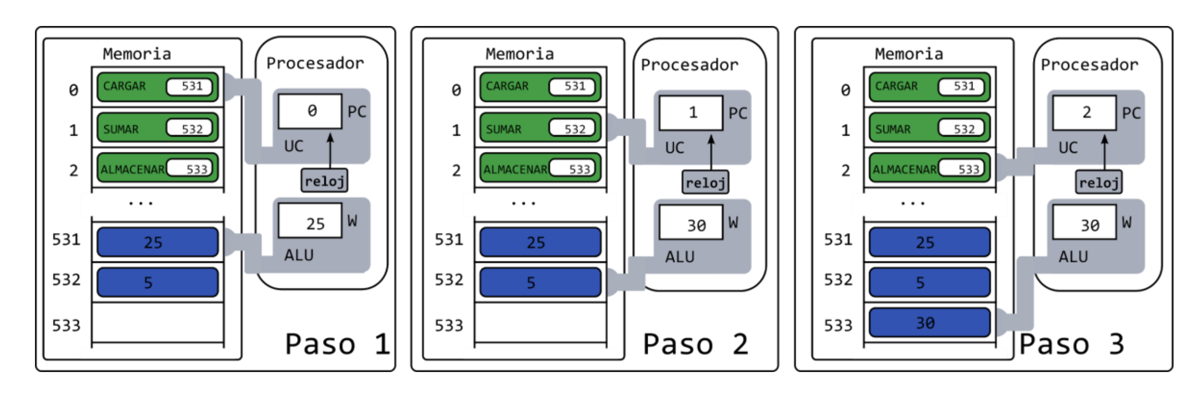


Figura 1.4. *Ejemplo simplificado de ejecución de código en un procesador*

A cada paso el contador de programa se incrementa y recoge una nueva instrucción que su circuitería ejecuta automáticamente. Cada vez que se incrementa PC, estamos ante un nuevo ciclo de instrucción y la velocidad a la que la computadora puede ejecutar estos ciclos de instrucción es un parámetro fundamental de los procesadores.

Al iniciar el procesador, en el Paso 1, PC tiene cargada la posición de memoria 0. Busca la instrucción en esa posición y encuentra "Cargar 531", por lo que eso es lo que realizan sus circuitos: buscan el código almacenado en la posición de memoria 531 y lo cargan en el registro W. Todo esto se lleva a cabo al producirse un pulso del reloj. Con el siguiente pulso, se incrementa PC de 0 a 1, en el Paso 2, y el procesador busca la instrucción en la dirección de memoria 1.

Esa instrucción es buscar lo que haya en la posición de memoria 532 y sumarlo a W. El procesador cumple esa tarea durante el pulso de reloj, y ahora en W se tiene la suma de 5 y 25. Finalmente, al pulsar nuevamente el reloj, PC se incrementa a 2, y en esa posición de memoria encuentra la instrucción "Almacenar 533" que indica que lo que está en W debe ser trasladado a la posición de memoria 533. Con el pulso continuo del reloj entonces, la ejecución del procesador ha logrado cumplir estas 3 instrucciones y efectuar una suma.

Las instrucciones en un procesador real por supuesto comprenden un conjunto mayor y más completo; no sólo pueden realizar operaciones aritméticas sino también lógicas; pero más importante, se convierte en una verdadera computadora y no una mera calculadora porque tiene instrucciones de saltos condicionales (puede tomar decisiones sobre qué camino seguir en función del resultado actual) e instrucciones repetitivas para implementar cualquier algoritmo, como veremos en detalle en el capítulo 2.

Entradas y salidas

El procesador necesita comunicarse con el mundo exterior, como vimos para acceder a las memorias, pero también para obtener o entregar datos de sus periféricos de entrada/salida. Comúnmente esta interacción ocurre a través de ciertas direcciones de memoria especiales donde los circuitos pueden depositar o extraer datos. Podemos imaginar una salida muy sencilla: por ejemplo, si se almacena el valor 1 en un registro especial se prende una luz y si se almacena un 0 se apaga. Un monitor puede tener un conjunto de 1920×1080 = 2.073.600 píxeles (cada píxel es un grupo de tres luces: roja, verde y azul para reproducir, combinándolas, cualquier color); con lo cual vemos que rápidamente los sistemas digitales se vuelven muy complejos. Aquí es donde podemos apreciar la ventaja de la computadora: si tuviéramos que configurar cada uno de los tres colores de los dos millones de píxeles, a mano, a razón de 1 por segundo, tardaríamos 72 días.

A una computadora cuya tasa de instrucción es, por ejemplo, de 1,5 GHz o mil quinientas millones de veces por segundo (algo fácilmente alcanzado por los celulares que portamos hoy en día), cada instrucción le toma menos de 1 nanosegundo. Incluso si necesitase 10 instrucciones para configurar cada píxel, resolvería toda la pantalla en 1/24 de segundo; suficiente para mostrar una película. Los procesadores modernos tienen además varios núcleos por lo que pueden resolver varios hilos de instrucciones en paralelo, y en realidad nos muestran imágenes valiéndose de procesadores especiales complementarios para manejar video y las complejas operaciones aritméticas de los juegos 3D; pero el ejemplo vale para demostrar cómo instrucciones sencillas ejecutadas a altas velocidades resultan en el comportamiento complejo y casi mágico que nos parece observar en las computadoras.

Las entradas, como un botón, se valen de circuitos especiales para producir una modificación en algún registro especialmente diseñado de la computadora al que el procesador puede acceder y leer su estado, conociendo así la voluntad de la persona que la utiliza. Versiones más sofisticadas de este esquema permiten el ingreso de datos por medio de teclados, *mouse*, comandos

de juegos, botoneras industriales, etc. Otro tipo de periféricos de entrada/salida manejan dispositivos que no están pensados para que una persona los utilice “manualmente” sino que conectan a la computadora con otros equipos electrónicos locales o remotos. Se denominan “puertos” y establecen una conexión electrónica para intercambiar información según diversos *protocolos*.

Complejidad y alto nivel: comprendiendo dónde programaremos

En las secciones anteriores se describieron las partes componentes del *hardware* de la computadora. También se describieron los comandos básicos que componen el set de instrucciones del procesador; ese es el nivel más bajo también del *software*, una computadora puede *programarse* colocando estas instrucciones en la memoria. El lenguaje de programación más básico de hecho se llama *assembler* y es simplemente un conjunto de etiquetas “amigables” para que una persona pueda escribir una secuencia del set de instrucciones del procesador y ejecutarla.

Las instrucciones de assembler sin embargo son relativamente simples y acotadas. Para crear un programa complejo necesitaríamos miles de instrucciones, lo cual es problemático no solo por el tiempo invertido, sino porque desarrollos tan extensos son proclives a errores y difíciles de comprender. Por lo tanto, se desarrollaron lenguajes de alto nivel, es decir, nuevos “sets de instrucciones” pero con sentencias más poderosas y fáciles de utilizar.

Algunos de los primeros fueron FORTRAN y COBOL. Más adelante se crearon muchos otros, en particular y de gran difusión el lenguaje C que aprenderemos en este libro. Una persona puede escribir un programa en estos lenguajes de más alto nivel y sus comandos se traducirán en aquellos más básicos del set de instrucciones del procesador para poder correr en la computadora. Esa traducción se conoce como *compilación* y la realiza a su vez otro programa, el compilador. Una ventaja muy importante de estos lenguajes es que el mismo código puede llevarse a distintas máquinas, aunque tengan procesadores con distintos sets de instrucciones, si tenemos el compilador apropiado para nuestro hardware.

Hoy en día, incluso C se ve como un lenguaje de relativamente bajo nivel y existen lenguajes como Python o JavaScript en un nivel de abstracción superior. Estas decenas de lenguajes distintos se utilizan para crear miles de programas o aplicaciones que utilizamos a diario, con funcionalidades muy diversas.

Por supuesto, en un principio, una computadora se encendía y corría un único programa que se había precargado en su memoria, como las antiguas consolas de videojuegos donde se introducía un “cartucho” por vez. Este programa contenía las instrucciones para administrar todos los aspectos del funcionamiento de la computadora. A medida que creció la complejidad de las computadoras y que a los usuarios les interesó utilizar diversos programas en diversas plataformas, este modelo de “programa único”, que además debía compilarse para cada computadora, debió abandonarse.

Pensemos en una computadora como una empresa: una empresa pequeña de producción de uniformes fundada por Jorge, su dueño y único empleado. Jorge debe saber diseñar y fabricar los uniformes, salir a venderlos a los comercios, llevar adelante la contabilidad. La

lista de tareas de Jorge dicta todo lo que pasa en la empresa y tiene detalles muy específicos. Ahora veamos lo que pasa luego de muchos años: Jorge es el CEO de un imperio comercial de producción de indumentaria. Sólo tiene que entrar a su oficina y lo que quiera está al alcance de su mano: el departamento contable le entrega los informes ya elaborados, los talleres producen lo que él les indica y puede pedir al departamento de diseño una actualización sobre los últimos modelos de la competencia. No sólo eso, sino que mientras Jorge duerme, su imperio sigue funcionando y muchos asuntos menores se resuelven a través de comunicaciones entre departamentos de las que Jorge no se entera nunca. La lista de tareas de Jorge ahora tiene unas pocas instrucciones de alto nivel, y ya no se preocupa por las máquinas de coser.

La gran empresa de Jorge es tan compleja como una computadora moderna, pero funciona gracias al análogo de lo que, en una computadora, llamamos *sistema operativo.* El sistema operativo (OS por sus siglas en inglés) es un conjunto de programas que se encargan de todas las funciones básicas de la computadora que siempre deben estar habilitadas al utilizarla y permiten que los programas de usuario se ejecuten en un entorno seguro y no se preocupen por características de bajo nivel. Algunos sistemas operativos muy difundidos son Windows, Android, iOS, diversas distribuciones de Linux, y Mac OS.

El límite que define exactamente qué programas corresponden al sistema operativo y cuales no puede ser un poco difuso, pero podemos demarcar con claridad los programas pensados específicamente para satisfacer una necesidad del usuario: escribir texto, ver páginas web, escuchar música, jugar juegos. Todos estos son programas de aplicación, "aplicaciones" o "*apps*". La figura 1.5 ilustra los componentes de la computadora, tanto de *hardware* como de *software*, descriptos hasta el momento.

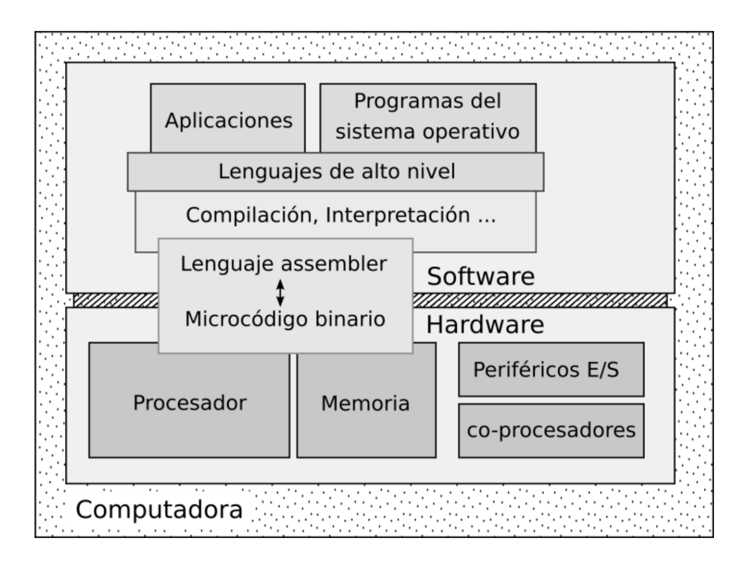


Figura 1.5. *Diagrama en bloques esquemático de partes componentes de la computadora*

En este libro se explicará la creación de estas aplicaciones en lenguaje C. Utilizaremos sentencias simples como "printf("Hola!");" y veremos aparecer en la pantalla un cartel que nos saluda, sin preocuparnos por aspectos de más bajo nivel. Sin embargo, una razón para

aprender el lenguaje C en particular, es que si bien nos permitirá crear estos programas que corren en un OS, también nos permitiría implementar programas que pueden correr en un microcontrolador sin la asistencia del OS. Esto está por fuera del alcance de este libro, pero es una habilidad importante en algunos campos de la ingeniería como la Electrónica, y la puerta de entrada para ello es aprender C.

Sistemas de numeración

Redescubriendo las bases

Utilizamos continuamente el *sistema de numeración decimal* sin pensar en ello, escribimos todos nuestros números en base 10, sabemos las tablas de multiplicación de memoria en esa base, es tan natural que ni siquiera pensamos que *existen otras alternativas*.

Antes de pasar a esas alternativas, pueden repasarse los conceptos básicos de la representación que usamos a diario. En la escuela primaria aprendimos a repetir: unidades, decenas, centenas; sabemos que un valor es una unidad y no una decena por la *posición* que ocupa en el número. Esto es porque, como la mayoría de las civilizaciones en la historia de la humanidad, usamos un *sistema posicional* de numeración. ¿Cómo sabemos cuánto vale el número 426,37 por ejemplo? Aplicamos la siguiente ecuación, sin pensarlo:

$$426{,}37 = 4 \times 10^2 + 2 \times 10^1 + 6 \times 10^0 + 3 \times 10^{-1} + 7 \times 10^{-2}$$

Es decir que cada dígito, además de tener un valor, tiene un peso asignado por su posición respecto de la coma. Ese peso está dado por la base de nuestro sistema de numeración, en este caso 10. Notemos algo más: el número 10 esconde más complejidad de la que aparenta; ahora que tenemos en mente las posiciones, podemos ver que en realidad 10 está compuesto por *dos dígitos*, el uno (1) y el cero (0) en dos posiciones distintas. $10 = 1 \times 10^1 + 0 \times 10^0$. Esto sucede porque un sistema base 10 tiene sólo 10 símbolos o dígitos (a los que estamos muy acostumbrados, el 0,1,2, … , 9); si empezamos a contar y llegamos al 9, encontramos que no hay más dígitos y debemos a recurrir a las *posiciones*, colocando un uno delante y recomenzando desde el 0 en las unidades.

En nuestra vida cotidiana, además de la base 10, utilizamos la base 12, generalmente para comprar huevos. Si pedimos “dos docenas” de huevos, ¿cuántos huevos estamos pidiendo? Podemos usar el mismo principio; estamos pidiendo dos docenas y cero unidades y por lo tanto lo escribimos

$$2 \times 12^1 + 0 \times 12^0 = 24.$$

Es decir que 24, en *base 12*, se escribe 20. En este punto tenemos que detenernos. El saber nos condena y ahora que sabemos que existen distintas bases y que 20 en base 12 vale 24 en base 10, ya no podemos mirar a cualquier número así como así y necesitamos saber en qué base está.

Para identificar cuál es la base *b* en que está escrito un número *n* se escribirá la base del número como subíndice tras un paréntesis: $n_{(b}$

Así, dos docenas se escribe $20_{(12}$y "veinte" se escribe $20_{(10}$. Por supuesto y como ya dijimos, $20_{(12} = 24_{(10}$.

El siguiente punto de interés quizás ya haya sido notado por un lector perspicaz: si en base 10 tengo 10 dígitos y al llegar al 9 tengo que recurrir a las posiciones y escribir$10_{(10}$, ¿qué sucede entre el $9_{(12}$ y el $10_{(12}$? Evidentemente, el sistema base 12 debe contar con 12 símbolos, no alcanzan los que conocemos entre 0 y 9, faltan 2. Afortunadamente hay fanáticos del sistema base 12 y la Sociedad Docenal de Gran Bretaña (¡Existe!) defendió con uñas y dientes la incorporación de los símbolos ↊y ↋para representar los símbolos faltantes (se pronuncian "dec" y "el"). Así, podemos contar en base 12:

0, 1, 2, … , 8, 9, ↊, ↋, $10_{(12}$, $11_{(12}$ … .

Por supuesto, ↊ vale $10_{(10}$ y ↋ vale $11_{(10}$.Existen tantas bases como números, pero algunas, que veremos en las siguientes secciones, son especialmente importantes para quienes programan y se usan mucho hoy en día, tanto que tienen nombres propios: se trata de las bases binarias (base 2) y la hexadecimal (base 16). La base 16 tiene el mismo problema que la base 12, en tanto que necesita más que los 10 símbolos del 0 al 9 que manejamos habitualmente, pero esto se resolvió sencillamente utilizando letras de la A a la F, de manera que $A_{(16} = 10_{(10}, B_{(16} = 11_{(10}, \ldots, F_{(16} = 15_{(10}$. La tabla 1.1 resume las bases y símbolos mencionados hasta el momento.

Antes de examinar las distintas bases, veremos cómo trabajar numéricamente con distintas bases.

Tabla 1.1. Ejemplos de sistemas de numeración más utilizados

Sistema de numeración	Base	Símbolos
Binario	2	0,1
Octal	8	0,1,2,3,4,5,6,7
Decimal	10	0,1,2,3,4,5,6,7,8,9
Docenal	12	0,1, 2,3,4,5,6,7,8,9,↊,↋
Hexadecimal	16	0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F

Teorema fundamental de la numeración

Entonces ahora tenemos distintas bases, y debemos asegurarnos de saber en cuál estamos trabajando antes de interpretar un conjunto de números. Y ni siquiera los dígitos son sagrados,

hemos abandonado el confort de los queridos numerales arábigos del 0 al 9 y hemos incorporado letras, entre otros símbolos. La interpretación de un número debe ser tratada con más formalidad.

Un sistema de numeración en base **b** utiliza un alfabeto compuesto por **b** símbolos o cifras para representar números. Los valores de los símbolos n_i cumplen $0 \leq n_i < \boldsymbol{b}$ y la representación del número dependen del valor de las cifras n_i y de las posiciones que las cifras ocupan dentro del número.

Puede obtenerse el valor en base 10 de un número **n** en base **b** a través del *teorema fundamental de la numeración*:

$$n_{(10} = n_N \ldots n_1 n_0, n_{-1} n_{-2} \ldots n_{-M\ (b}$$

$$= n_N \times b^N + \ldots + n_1 \times b^1 + n_0 \times b^0 + n_{-1} \times b^{-1} + n_{-2} \times b^{-2} + \ldots + n_{-M} \times b^{-M}$$

Por ejemplo: $324{,}121_{(5} = 3 \times 5^2 + 2 \times 5^1 + 4 \times 5^0 + 1 \times 5^{-1} + 2 \times 5^{-2} + 1 \times 5^{-3} = 89{,}288_{(10}$

Conversión entre bases

La base 10 es particular porque es la que aprendimos desde la infancia y entendemos los números con sólo verlos. No solo eso, sino que es la base en la que sabemos hacer cuentas. Por ello, casi siempre para convertir un número de una base a otra es necesario pasar por la base 10. Analizaremos los métodos para hacerlo. Ya hemos aprendido uno de ellos:

Para convertir entre cualquier base y base 10 puede usarse el teorema fundamental de la numeración

---

En el caso particular del código binario el teorema fundamental adquiere una forma particularmente sencilla porque cada dígito tiene un peso que es simplemente el doble del anterior. Por ejemplo, para un número de 8 bits se tiene:

$$n = n_7 \times 2^7 + n_6 \times 2^6 + n_5 \times 2^5 + n_4 \times 2^4 + n_3 \times 2^3 + n_2 \times 2^2 + n_1 \times 2^1 + n_0 \times 2^0$$

$$= n_7 \times 128 + n_6 \times 64 + n_5 \times 32 + n_4 \times 16 + n_3 \times 8 + n_2 \times 4 + n_1 \times 2 + n_0 \times 1$$

Quienes programan habitualmente, directamente recuerdan o deducen estos valores en el momento y suman esos “pesos” según cada bit esté “prendido” o “apagado”.

Por ejemplo, podemos calcular el valor de $10\ 0110_{(2}$ con el siguiente proceso:

Número binario	1	0	0	1	1	0 (2
Peso de cada bit	2^5	2^4	2^3	2^2	2^1	2^0
	32	16	8	4	2	1

➡

1	0	0	1	1	0 (2
32	16	8	4	2	1

➡ $32+4+2 = 38_{(10}$

Para convertir “en el otro sentido”, de base 10 a otra base, podemos usar el siguiente procedimiento:

Para convertir un número de base 10 a cualquier otra base debemos separar la parte entera de la parte fraccional y aplicar un procedimiento distinto a cada parte.

Notemos que para cualquier base *b*, el dígito que está al lado de la coma siempre estará pesado por el factor b^0que vale 1 sin importar cuál sea la base b (cualquier número elevado a 0 da 1). Por lo tanto, la coma es un punto "inmutable" al transformar de base en base y podemos trabajar siempre con la parte entera por separado de la parte fraccionaria.

Ahora veamos cómo convertir la parte entera.

Para convertir un número *n* entero de base 10 a cualquier base *b*, realizamos los siguientes pasos del proceso conocido como *división continua*:

1. Realizar la **división entera** de *n* por *b* para obtener el cociente *c* y el resto r_0
2. Repetir el paso anterior tomando el cociente *c* como el nuevo *n* obteniendo un nuevo cociente y el resto r_i
3. Repetir el paso 2 hasta que el cociente sea cero obteniendo el resto r_m
4. Si es necesario traducir los restos r_i, que están en base 10, a los símbolos equivalentes en base *b* (esto es necesario cuando *b* > 10)
5. El número en base *b* es $r_m\ r_{m-1} \dots\ r_1\, r_0$

Por ejemplo, obtengamos la representación en base 16 del número $2862_{(10}$. El proceso de división contínua suele escribirse repitiendo cada división directamente al lado del cociente anterior, resultando un esquema como el siguiente:

Como puede verse, los restos que se obtuvieron son 11, 2 y 14, que deben luego escribirse en ese orden: comenzando por el resto de la última división hacia el primero. Pero cuidado, porque estos números están en base 10. Debemos obtener los símbolos equivalentes en base 16 para poder escribir un número en esa base. Recordando la tabla, obtenemos B, 2, E y entonces se concluye que $2862_{(10} = B2E_{(16}$. Es interesante notar que la última división da 0, y si siguiéramos realizando divisiones enteras seguiríamos obteniendo 0 con resto 0; por lo tanto, si incorporáramos esos valores al número seguiría siendo correcto, los ceros a la izquierda no cambian el valor de un número.

Los números binarios son quizás una excepción ya que, si bien puede obtenerse la representación base 2 con el método de división continua, es muchas veces más cómodo considerar cuáles bits "prender o apagar" para obtener el valor deseado.

Por ejemplo, si queremos pasar $198_{(10}$a base 2, podemos recordar que 2^8 es 256 y 2^7 es 128. Por lo tanto, desde el bit que pesa 2^8 (el bit 9) y subiendo, todos deben estar "apagados", de otra manera nos pasaríamos. Si se "enciende" el bit 8 se tiene 128 y el próximo bit que puede encenderse es el bit 7 (pesa 2^6=64) para llegar a 192. Falta sumar 6 para llegar a 198, que obtenemos prendiendo el bit 3 (2^2=4) y el 2 (2^1=2). Si realizamos este proceso en una hoja se vería de esta manera:

		✗	✗	✓	✓	✗	✗	✗	✓	✓	✗
		512	256	128	64	32	16	8	4	2	1
		2^9	2^8	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
$198_{(10}$	=	0	0	1	1	0	0	0	1	1	$0_{(2}$

Ahora bien, habiendo resuelto el pasaje de la parte entera de los números, debemos resolver el pasaje de la parte fraccionaria. Para entender la solución que presentaremos, recordemos que un número fraccionario *n* en base *b* se escribe en base 10 como $n = n_{-1} \times b^{-1} + n_{-2} \times b^{-2} + \ldots$ por el teorema fundamental de la numeración. Observemos lo que pasa cuando multiplicamos este número fraccionario por la base:

$$b \times n_{(b} = b \times (0, n_{-1}n_{-2}n_{-3} \ldots) = b \times (n_{-1} \times b^{-1} + n_{-2} \times b^{-2} + n_{-3} \times b^{-3} + \cdots)$$

$$= n_{-1} \times b^{0} + n_{-2} \times b^{-1} + n_{-3} \times b^{-2} + \ldots = n_{-1}, n_{-2}n_{-3} \ldots$$

El coeficiente que era multiplicado por b^{-1} pasa a estar del lado de los enteros, a la izquierda de la coma. Lo interesante es que así se demuestra que no importa en qué base escribamos el número, al multiplicar por *b* "aparece" el coeficiente n_{-1}del lado de los enteros de manera que

$$b \times n_{(b} = b \times 0, n_{-1}n_{-2}n_{-3} \ldots = \boldsymbol{n_{-1}}, n_{-2}n_{-3} \ldots$$

$$b \times n_{(10} = b \times 0, x_a x_b x_c \ldots = \boldsymbol{n_{-1}}, x'_a x'_b x'_c \ldots$$

Por ejemplo, tomemos el $0{,}423_{(6}$ que vale $0{,}7361_{(10}$. Multiplicando cualquiera de sus representaciones por la base 6 se tiene:

$$6 \times 0{,}423_{(6} = \mathbf{4}, 23_{(6}$$

$$6 \times 0{,}7361_{(10} = \mathbf{4}, 616_{(10}$$

Demostrando la obtención de $n_{-1\,(6}$ como 4. Esto es muy ventajoso, porque nos permite identificar el coeficiente que efectivamente corresponde al número en base *b* aunque estemos trabajando en base 10. Más aún, como sabemos que podemos trabajar en la parte entera y fraccionaria de los números por separado, ahora podemos retirar la parte entera, quedarnos con la fraccionaria, y repetir el procedimiento para obtener $n_{-2\,(6}$. El método completo se enuncia como sigue:

Para convertir un número fraccionario de base 10 a la base b debemos multiplicar el número por la base b, obtener la parte entera e_0, eliminar esa parte entera, y continuar el proceso repetitivamente obteniendo las partes enteras e_i. Finalmente, el número es $0, e_0 e_1 \ldots e_m$

Por ejemplo, convirtamos $0{,}6875$ a base 2:

$$0{,}6875 \times 2 = 1{,}375 \Rightarrow e_0 = 1$$
$$0{,}375 \times 2 = 0{,}75 \Rightarrow e_1 = 0$$
$$0{,}75 \times 2 = 1{,}5 \Rightarrow e_2 = 1$$
$$0{,}5 \times 2 = 1{,}0 \Rightarrow e_2 = 1$$

Como las próximas multiplicaciones dan 0, nos detenemos aquí y resulta:

$$0{,}6875 = 0, e_0 e_1 e_2 e_3 {}_{(2} = 0{,}1011_{(2}$$

El ejemplo fue engañosamente seleccionado para ser sencillo. No siempre el proceso termina limpiamente en 0, ya que muchas veces los números que queremos encontrar tienen infinitas cifras fraccionarias. En ese caso, nos veremos obligados a detenernos en algún punto y acotar el *error* que estemos cometiendo por no incluir a las demás cifras. Si nos detenemos en la cifra n_{-i}, el error entre el número real y el que calculamos es:

$e = (n_{-1} \times b^{-1} + \ldots + n_{-i} \times b^{-i} + n_{-i-1} \times b^{-i-1} + \ldots)$- $(n_{-1} \times b^{-1} + \ldots + n_{-i} \times b^{-i})$= $n_{-i-1} \times b^{-i-1} + \ldots$

El máximo valor que puede tomar e se da si todos los coeficientes n_{-i-1}alcanzan su máximo valor, y puede demostrarse[7] que la suma de todos esos valores es menor o igual al peso del coeficiente anterior, es decir b^{-i}. Por lo tanto:

Si se trunca un número en base b en i dígitos fraccionarios, se comete un error menor o igual a b^{-i}

Veamos el caso del número $n_{(10} = 0,4_{(10}$, si aplicamos el proceso anterior obtenemos que puede escribirse en base 2 como $n_{(2} = 0{,}0110011001100\ldots_{(2}$. Evidentemente no pueden escribirse los infinitos dígitos de este valor, por lo tanto, decidimos detenernos luego de 6 dígitos de manera que $\bar{n}_{(2} = 0{,}011001$. Para calcular el error podemos volver a la base 10 y obtener

$$e = n_{(10} - \bar{n}_{(10} = 0,4_{(10} - 0.390625_{(10} = 0.009375\,,$$

que efectivamente es menor a $2^{-6} = 0.015625$ (pero no menor a $2^{-7} = 0.0078125$).

Hemos visto por separado los procedimientos para convertir la parte entera de un número a cualquier base, y por otro lado la parte fraccionaria. Para convertir un número completo simplemente se reúnen ambas partes luego de convertirlas por separado.

Bases importantes en programación

Llegamos al punto del capítulo que explica por qué en este libro estamos hablando de sistemas de numeración y bases. Existen dos bases que son muy importantes en el contexto de la programación. La primera es relativamente “famosa” y es la base 2 o *base binaria*. Es importante porque es todo lo que la computadora “conoce”. Se asocian los niveles “bajo” y “alto” que los

7 Por la sumatoria geométrica que resulta

circuitos digitales pueden representar a los dos símbolos del sistema de numeración base 2: 0 y 1, que son los valores que puede tomar un bit.

Quienes trabajan programando a más bajo nivel sin duda necesitan utilizar directamente números en base 2 (llamados números binarios). Quienes programan a más alto nivel también necesitan conocer los detalles del código binario porque, como veremos más adelante, toda la información se maneja de esta manera en la computadora y el hecho de que todo se convierta a base 2 tiene un impacto que es necesario considerar.

Si esto no parece convincente, examinemos lo que pasa con el inocente número $0{,}2_{(10}$al querer representarlo en base 2. Utilizando el método visto en las secciones anteriores encontramos que se representa como $0{,}001100110011\ldots_{(2}$ es decir que, en base 2, es periódico y tiene infinitos dígitos luego de la coma. Por lo tanto, en la práctica será muy difícil representarlo en forma exacta en código binario (ningún sistema tiene infinitos bits), a menos que sepamos en qué sistema es posible.

Debido a las características de la arquitectura de las computadoras, es conveniente agrupar los bits en conjuntos de 8 llamados *bytes*. Las primeras computadoras estaban construidas tomando como unidad básica a los bloques de 1 byte[8]. Esta agrupación en bloques de 8 bits fue una tecnología sobre la que se construyeron todos los dispositivos digitales que evolucionaron hasta los que tenemos hoy, y por lo tanto impactó en muchas tecnologías.

Por ejemplo, los colores que vemos en una página web se definen con un protocolo RGB (de las iniciales en inglés para Rojo, Verde, Azul) donde se indica la intensidad de rojo, verde y azul con 3 valores entre 0 y 255. El rango entre 0 y 255 abarca 256 valores que son los que se pueden representar con 8 bits (8 casilleros con 2 posibilidades cada uno resulta en $2^8 = 256$combinaciones). Así, el color rojo puro es (11111111, 00000000, 00000000) y el azul (00000000,00000000,111111111).

Escribir el valor de los bytes usando números binarios es tedioso y proclive a errores, por lo tanto, en su lugar se usó y se usa muy comúnmente la base 16 (hexadecimal). ¿Por qué esta base y no la base 10? La base 16 tiene la particularidad de que cada posición puede tomar un valor entre $16 = 2^4$ valores, por lo cual cada dígito hexadecimal puede ser representado *exactamente* por un conjunto de 4 bits y viceversa. Por lo tanto, no se necesitan métodos complejos para pasar de un sistema a otro, la conversión entre los dos sistemas es directa.

Lo mismo sucede con el sistema octal ya que $8 = 2^3$ y cada dígito octal puede representarse con 3 bits. Sin embargo, como los bits se agrupaban de a 8, era más cómodo el sistema hexadecimal ya que con dos dígitos "hexa" se representaba exactamente 1 byte completo. La equivalencia entre valores de las bases mencionadas se muestra en la tabla 1.2.

Puede observarse fácilmente en la tabla 1.2 que el valor no es más ni menos que la conversión a base 10 del número binario o hexadecimal usando el teorema fundamental de la numeración. Así, el color "verde bosque" que se escribiría en (R,G,B) binario como:

---

[8] Esto es usa sobre simplificación, ha habido sistemas de distinto número de bits, pero la norma fue siempre conjuntos de 8. Luego de los primeros de 1 y 2 bytes, hoy en día los sistemas comúnmente tienen bloques 64 bits (4 bytes), aunque aún quedan muchos de 32 bits.

$$(00100010, 10001011, 00100010)$$

puede traducirse en:

(0x22, 0x8B, 0x22).

Tabla 1.2. Valores del 1 al 16 expresados en distintas bases (se muestra en dos mitades continuadas).

Binario	Hexa	Octal	Valor	Binario (cont)	Hexa (cont.)	Octal (cont.)	Valor (cont.)
0000	0	0	0	1000	8	10	8
0001	1	1	1	1001	9	11	9
0010	2	2	2	1010	A	12	10
0011	3	3	3	1011	B	13	11
0100	4	4	4	1100	C	14	12
0101	5	5	5	1101	D	15	13
0110	6	6	6	1110	E	16	14
0111	7	7	7	1111	F	17	15

De la misma manera, un número binario largo como 1100001011111001 podría representarse de manera más compacta con un número base 10: 49913. Sin embargo, la conversión no es directa; en cambio, el pasaje a base 16 se puede realizar directamente agrupando de a 4 bits:

$$1100\ 0010\ 1111\ 1001_{(2}$$

$$C\quad 2\quad F\quad 9_{(16}$$

Después de un tiempo de usar esta herramienta, la conversión es rápida y se puede hacer “de memoria”. Esta forma de escribir los números binarios no sólo es más corta, sino que hace mucho más fácil detectar y evitar errores. Quienes programan a bajo nivel, generalmente micro-controladores, la utilizan asiduamente.

Aritmética en bases distintas de la decimal

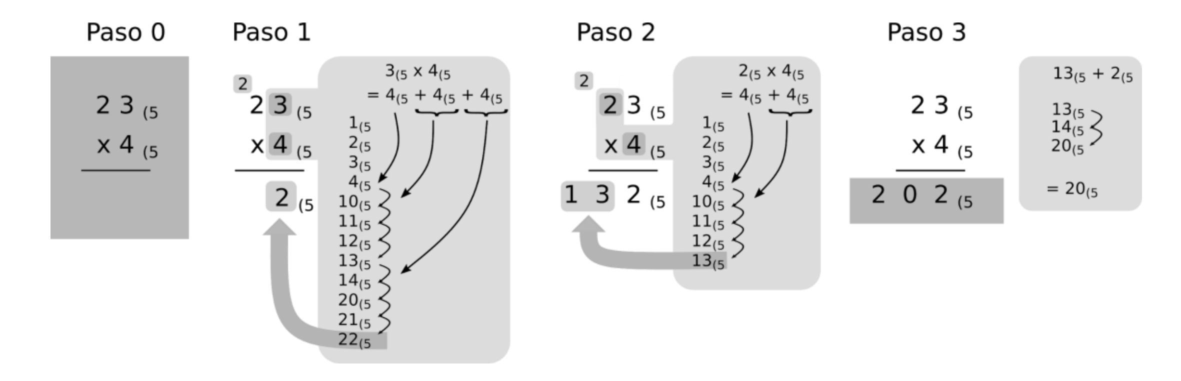
Si bien es válido expresar un número en cualquier base, hemos aprendido desde nuestra infancia a realizar operaciones aritméticas en base 10 y no en otra base, específicamente porque memorizamos las tablas de multiplicar en base 10 y tenemos práctica con las sumas también en base 10. No es difícil obtener el siguiente resultado “de memoria”: $7_{(10} + 6_{(10} = 13_{(10}$, pero no podemos hacer cuentas en forma directa tan fácilmente en este caso: $7_{(8} + 6_{(8}$. Para sumarle 6 unidades a 7 en base 8 debemos recordar que los símbolos de base 8 van del 0 al 7; cuando en base 10 se

agotan los dígitos, recurrimos a agregar un número en la posición de la izquierda como ya se discutió; así $9_{(10} + 1_{(10} = 10_{(10}$. Análogamente en base 8 se tiene $7_{(8} + 1_{(8} = 10_{(8}$. Por lo tanto, para resolver $7_{(8} + 6_{(8}$ podemos pensarlo como $7_{(8} + 6_{(8} = 7_{(8} + 1_{(8} + 5_{(8} = 10_{(8} + 5_{(8} = 15_{(8}$. Podemos comprobar nuestros resultados utilizando dos caminos distintos y comparando:

$$15_{(8} = (\, 1 \times 8^1 + 5 \times 8^0) \ = 13_{(10}$$

$$7_{(8} + 6_{(8} \ = \ (\, 7 \times 8^0) \ + \ (\, 6 \times 8^0 \,) \ = 13_{(10}$$

Similarmente, si queremos resolver $23_{(5} \times 4_{(5}$debemos volver a la escuela primaria y escribir la operación completa paso a paso:



En el paso 0 se plantea la multiplicación como es tradicional en el método escrito, recordando que en este caso todos los valores están en base 5. En cada paso subsiguiente se muestran las operaciones auxiliares. En este caso se realizan las operaciones directamente en base 5 utilizando un método gráfico para la suma ya que no sabemos realizar de memoria sumas en esa base. Otra alternativa hubiese sido pasar la cuenta a base 10, realizar la operación, y volver a base 5. Por ejemplo, en el paso 1:

$3_{(5} \times 4_{(5} \equiv \ 3_{(10} \times 4_{(10} = \ 12_{(10}$, luego se utiliza división continua para volver a base 5

$$\begin{array}{rl} 12 & \lfloor\ 5 \\ -\,10 & \ \ 2 \ \lfloor\ 5 \\ 2 & -0 \quad 0 \\ & \ \ 2 \end{array}$$

y se obtiene $22_{(5}$. Al finalizar el proceso completo se obtiene el resultado $23_{(5} \ \times 4_{(5} = \ 202_{(5}$, el cual puede comprobarse realizando las operaciones por caminos distintos:

$$23_{(5} \times 4_{(5} = 202_{(5} = 2 \times 5^2 + 0 \times 5^1 + 2 \times 5^0 = 52_{(10}$$

$$23_{(5} \times 4_{(5} = (2 \times 5^1 + 3 \times 5^0) \times (4 \times 5^0) \ = \ 13_{(10} \times 4_{(10} = 52_{(10}$$

Realizar cálculos en otras bases es en general complicado, y puede ser más fácil pasar los valores a base 10, luego realizar la cuenta en la base 10 que conocemos, y finalmente convertir de nuevo el resultado a la base de interés. Por supuesto, en sistema binario y en un sistema que utilizamos asiduamente como el hexadecimal, podemos acostumbrarnos a las operaciones sencillas como sumas y restas, y puede incluso resultar más cómodo evitar volver a la base 10. Observemos una misma operación de suma en estas bases:

$$\begin{array}{r} {}^{1}13_{(10} \\ +\ \ 7_{(10} \\ \hline 20_{(16} \end{array} \qquad \begin{array}{r} {}^{1}1\,{}^{1}1\,{}^{1}0\ 1_{(2} \\ +\ \ 0\ 1\ 1\ 1_{(2} \\ \hline 1\ 0\ 1\ 0\ 0_{(2} \end{array} \qquad \begin{array}{r} D_{(16} \\ +\ \ 7_{(16} \\ \hline 14_{(16} \end{array}$$

En los tres casos se realiza la misma operación con los mismos números, pero con distintas bases. La clave para analizar estas sumas es recordar que mientras $9_{(10} + 1_{(10} = 10_{(10}$, en base binaria $1_{(2} + 1_{(2} = 10_{(2}$ y en hexadecimal $F_{(16} + 1_{(16} = 10_{(16}$.

Sistemas de representación

Representación de la información en la computadora

"Representar información" es simplemente el acto de convertir cualquier tipo de dato que queramos guardar o manejar en una computadora a un formato binario. Números, texto, imágenes, video, todo eso que utilizamos a diario en una computadora sólo puede existir en ella como una colección de unos y ceros. Si de alguna manera lográramos observar un pequeño bloque de la memoria de la computadora, por ejemplo una computadora de 32 bits, veríamos lo siguiente:

`1000 1100 1110 0111 0101 1010 0011 1111`

Allí, en ese mar de 1s y 0s, en realidad podemos estar observando letras, números o un pequeño fragmento de una imagen. Entonces, ¿cómo sabemos qué estamos viendo? Para interpretar esa información necesitamos una convención, aplicar un molde o formato que permita guardarla y recuperarla, interpretando siempre lo mismo. En las siguientes secciones veremos las convenciones más usadas existentes, qué información puede representarse con ellas, y cuáles son sus limitaciones. En particular nos enfocaremos en:

- Sistemas de representación de números enteros
  - Binario puro
  - Módulo y signo
  - Complemento a 1
  - Complemento a 2
  - Exceso
  - Decimal codificado en binario
- Sistemas de representación de números reales
  - Punto fijo
  - Punto flotante IEEE754
- Sistemas de representación de caracteres
  - ASCII

Dedicaremos especial atención a la manera en que se realizan operaciones matemáticas en estos sistemas de representación. Para facilitar la notación, de la misma manera que utilizamos el subtexto con paréntesis para indicar la base, como en $110_{(2}$, utilizaremos el corchete para indicar el sistema de representación cuando sea posible, por ejemplo $0110_{[BP4}$. En general, los

números binarios se escribirán separados en grupos de 4 para facilitar la lectura, como es tradicional por la ya comentada equivalencia con dígitos hexadecimales.

Representación de números enteros

Sistema "Binario puro"

La representación binaria pura (BP) permite representar números enteros positivos. El único parámetro en el que debemos ponernos de acuerdo es la cantidad de bits que usaremos para representar cada número. Hablamos siempre de un sistema de representación binario puro de n bits. Por ejemplo, puede establecerse que cada número se representará utilizando $n = 8$ bits, y entonces podremos determinar que la representación del número 62 es su conversión a la base binaria, completando con ceros a la izquierda hasta llenar los 8 "casilleros" disponibles.

$62_{(10} = 111110_{(2}$ ⇒Representación en binario puro de 8 bits: `0011 1110`

Este formato permite representar 2^n valores. El menor número que puede representarse en este formato se obtiene con todos sus bits en cero, que representa al cero (parece obvio, pero no siempre será así en otros sistemas). El mayor número se alcanza cuando todos sus bits están en uno, valor equivalente a $2^n - 1$. Para el caso de $n = 8$ sería $1111\ 1111$que representa al $255_{(10}$.

Este sistema sencillo ya demuestra los problemas característicos de los sistemas de representación: al contar con n bits (un tamaño finito) las operaciones matemáticas no siempre dan un resultado que quepa en el rango disponible. La cuenta $74_{(10} + 223_{(10}$en un sistema binario puro de 8 bits resulta:

`0111 0100 + 1101 1111 = **1** 0101 0011`

Claramente ocurrió un acarreo (o *carry* por nombre en inglés) hacia el noveno bit de la izquierda, pero este bit no existe en un sistema de 8 bits. Esta condición se conoce como *overflow* o desborde y el resultado es, en la mayoría de los casos, que ese bit de acarreo se pierde produciéndose el fenómeno de *wrap* o envoltura, porque la apariencia de lo sucedido es como si se hubiese continuado sumando desde 0. Para ilustrar claramente este fenómeno veamos qué pasa si sumamos de a uno desde el anteúltimo valor en binario puro de 8 bits:

Binario Puro	Valor	Binario Puro 8 bits
1111 1110	$254_{(10}$	$254_{[BP8}$
1111 1111	$255_{(10}$	$255_{[BP8}$
1 0000 0000	$256_{(10}$	$0_{[BP8}$
1 0000 0001	$257_{(10}$	$1_{[BP8}$

Luego de “llenar” los 8 bits menos significativos ocurre un acarreo y como se ve en la columna de la derecha, para el sistema binario puro de 8 bits, que no reconoce el noveno bit, los valores recomienzan desde el primero.

Sistema “Módulo y signo”

La representación en módulo y signo (MyS) permite representar números enteros con signo. Para ello también se define un tamaño n, pero el bit más significativo se reserva para indicar el signo: 0 significa positivo y 1 negativo. Los $n-1$ bits restantes indican el valor absoluto del módulo.

Si se quiere representar el $-5_{(10}$ y $5_{(10}$ en módulo y signo se obtiene:

Valor $-5_{(10}$	Número binario: $-101_{(2}$	MyS 8 bits:	**1000 0101**
Valor $-5_{(10}$	Número binario: $-101_{(2}$	MyS 16 bits:	**1000 0000 0000 0101**
Valor $5_{(10}$	Número binario: $101_{(2}$	MyS 8 bits:	**0000 0101**

Es importante notar que en este caso existen dos valores que representan al 0, con 8 bits serían el `0000 0000` y el `1000 0000`. Es decir que existe el 0 y el -0. Esto es una desventaja del sistema, ya que la computadora debe chequear dos códigos distintos para saber si el valor es cero. El máximo número positivo que se puede representar en este caso es $2^{n-1}-1$, ya que comparando con el binario puro hemos quitado un bit para el signo. El valor negativo de mayor módulo es similarmente $-(2^{n-1}-1)$y se dice que el rango es *simétrico* porque el máximo módulo positivo es igual al negativo.

Para este sistema, el desborde con efecto de envolvente resulta en un cambio de signo.

Sistema “Complemento a 1”

Este sistema[9] permite representar enteros con signo. Se define un tamaño n y luego se siguen las siguientes reglas: si el número es positivo se representa en binario puro, y si es negativo se representa aplicando la operación de *complemento a 1* (Ca1) que no es más que invertir todos sus valores, es decir, cambiar los 1s por 0s y los 0s por 1s. Esta operación de inversión es equivalente a restar el número a 2^n-1.

Las representaciones de $-5_{(10}$y $5_{(10}$resultan:

Valor $-5_{(10} = -101_{(2}$	Ca1 8 bits:	*paso 1: acomodar en 8 bits*	0000 0101
		paso 2: es negativo, invertir	**1111 1010**
Valor $5_{(10} = 101_{(2}$	Ca1 8 bits:	*paso 1: acomodar en 8 bits*	0000 0101
		paso 2: es positivo, no invertir	**0000 0101**

[9] El nombre “complemento a 1” se aplica al sistema de representación, pero también existe la operación de “complemento a 1”, lo cual resulta un tanto confuso. Se intentará clarificar a qué se está haciendo referencia en cada caso.

Como característica importante, la representación de todos los números negativos tiene un 1 en su bit más significativo. Pero debido a esto, el número positivo más grande que puede representarse es $2^{n-1} - 1$como en el caso de módulo y signo. Es importante notar que en módulo y signo se "reserva" un bit para el signo, pero en Ca1 no ocurre lo mismo y no hay que hacer nada especial con el bit más significativo, sino que simplemente se limita el mayor número positivo y al invertir los negativos, el último bit resulta ser 1.

El rango de este sistema también es simétrico de$-(2^{n-1} - 1)$a $2^{n-1} - 1$.

Observemos qué pasa en Ca1 de 8 bits con el código `1111 1111`. Como tiene un 1 adelante, debe representar un número negativo; averiguamos su módulo invirtiéndolo, lo cual nos da `0000 0000`, por lo tanto, representa -0. Otra vez, tenemos dos representaciones para el cero.

Sistema "Complemento a 2"

El sistema complemento a 2 permite representar números enteros con signo y es el más utilizado actualmente. En forma similar al Ca1, se establece un número de bits n y si el número es positivo se representa en binario puro, mientras que si es negativo se representa según su "*complemento a 2"* (Ca2). La operación de complemento a 2 involucra invertir todos los valores y sumar 1.

Las representaciones de $-5_{(10}$y $5_{(10}$resultan:

Valor $-5_{(10} = -101_{(2}$	Ca2 8 bits:	*paso 1: acomodar en 8 bits*	`0000 0101`
		paso 2: es negativo, invertir	`1111 1010`
		paso 3: sumar 1	**`1111 1011`**
Valor $5_{(10} = 101_{(2}$	Ca2 8 bits:	*paso 1: acomodar en 8 bits*	`0000 0101`
		paso 2: es positivo, no invertir	**`0000 0101`**

Notemos que la operación de complemento a 2 es equivalente a restar el número a 2^n. Por ejemplo, para el número n_1 en Ca2 de 8 bits:

(invertir(n_1))+1 = (1111 1111 - n_1) +1 = (1111 1111 + 1) - n_1 = 1 0000 0000 - n_1 = $2^n - n_1$

Un procedimiento alternativo para obtener el Ca2 es buscar el primer 1 en su representación de binario puro de la derecha a la izquierda, dejar ese 1 sin modificar, e invertir el resto hacia la izquierda. Si se quiere aplicar esta regla al $-52_{(10}$por ejemplo obtenemos el binario puro de su módulo 00110100, buscamos el primer 1 desde la derecha, marcado a continuación en negrita: 0011 0**1**00, lo dejamos como está y luego invertimos el resto: **1100 1**100.

De la misma forma que en el caso del Ca1, los números negativos tienen un 1 en la posición más significativa.

Para el procedimiento inverso, es decir, obtener el valor en base diez a partir de la representación en Ca2, si el número comienza en 0 se interpreta como binario puro. Si comienza en 1 y por lo tanto representa un negativo, debe "des-complementarse a 2" y para eso hay distintos métodos matemáticamente equivalentes que pueden seguirse. El primero es realizar exactamente la operación inversa: restar 1 y luego invertir todos los dígitos. El segundo, quizás más sencillo, es repetir la operación de complemento a 2: invertir y sumar 1 (o el procedimiento alternativo de buscar el

primer 1 desde la derecha). En estos dos casos obtenemos un número positivo y debemos *recordar* que el número era negativo para “agregarle” el menos luego de la conversión. El tercer método arroja un resultado negativo directamente: consiste en considerar que el peso del bit más significativo es negativo y usar el teorema fundamental de la numeración con esa “modificación”. Veamos un ejemplo con esas 3 variantes donde explicitaremos su igualdad matemática:

Dado $n_1 = -26_{(10}$con módulo en binario puro 0001 1010 y representación Ca2 de 8 bits `1110 0110`,
para volver a obtener $-26_{(10}$a partir `1110 0110` realizamos:

Opción 1: la operación inversa. Consiste en realizar la inversión de (1110 0110 - 0000 0001). Sin dudas nos da el resultado buscado ya que es retroceder en los pasos que nos condujeron hasta aquí. Efectivamente: 1110 0110-1 = 1110 0101 que invertido es 00011010 y por lo tanto el valor original es -00011010 = $-26_{(10}$.

Opción 2: Pero, invertir $n_1$es equivalente a efectuar la resta 1111 1111 - n_1, es decir que la operación anterior puede ser escrita matemáticamente como:

1111 1111 - (1110 0110 - 0000 0001) = 1111 1111 - 1110 0110 + 0000 0001

que es igual a invertir 1110 0110 y luego sumarle 1, es decir a complementar a 2 nuevamente el número. Al terminar la operación, debemos agregar el menos.

Opción 3: Finalmente expresamos matemáticamente el tercer método:
n_1 = - 1000 0000 + 0110 0110 = -(0111 1111 + 1) + 0110 0110 = - (0111 1111 - 0110 0110 +1) que es exactamente igual a la opción 2, pero sin la necesidad de agregar el menos.

La representación en complemento a 2 tiene dos diferencias importantes respecto a las anteriores. En primer lugar, existe un sólo código para el 0, que en 8 bits es `0000 0000`. En segundo lugar, el rango de representación *no es simétrico*. El sistema Ca2 de n bits tiene un rango de -2^{n-1}a $2^{n-1} - 1$. Es decir que aloja un valor negativo más que valores positivos.

El sistema de representación Ca2 es el más utilizado en todos los sistemas de cómputo debido a que simplifica ampliamente las operaciones matemáticas. Si dos números están representados en Ca2, pueden sumarse o restarse siguiendo los mismos procedimientos *sin importar si son positivos o negativos* con la salvedad de que *debe descartarse el carry-over.* En otras palabras, no es necesario convertir los números a binario puro antes de hacer restas o sumas. Imaginemos que tenemos un número n_1 positivo al que queremos restar uno negativo n_2, ambos en Ca2. Esto resulta:

$$n_{1\,[Ca2} - n_{2\,[Ca2} = n_{1\,[BP} - (2^n - 1 - |n_2| + 1) = n_{1\,[BP} + |n_2| + 2^n$$
$$= n_{1\,[BP} + n_{2\,[BP} + 2^n$$

Como se ve, si se descarta 2^n, que es justamente el acarreo, el resultado es correcto.

En este contexto parece difícil detectar cuándo ocurrió un desborde. Para eso, se necesita observar el proceso de suma o resta y verificar qué ocurre con los acarreos o “préstamos” de los dos bits más significativos. La regla dicta que si los dos son iguales, no ocurrió un desborde.

Examinemos ejemplos de esta regla al realizar diversas operaciones con los números de módulo $122_{(10}$y $100_{(10}$en la tabla 1.3. Los casos donde ocurre un desborde son los que exceden el rango para Ca2 8 bits [-128, 127] y se indican con una cruz.

Tabla 1.3. Operaciones de suma (con sumandos S1 y S2) y resta (con minuendo R1 y sustraendo R2) y sus resultados (R), con una fila extra explicitando los acarreos (A) o préstamos (P).

122+100=222 ✖	100-122=-22 ✓	-100+-122=-222 ✖	-100+122=22 ✓
A: **01**100 000	P: **11**111 101	A: **10**011 100	A: **00**111 000
S1: 0111 1010	R1: 0110 0100	S1: 1001 1100	S1: 1001 1100
S2: 0110 0100	R2: 0111 1010	S2: 1000 0110	S2: 0111 1010
R: 1101 1110	R: 1110 1010	R: 0010 0010	R: 1001 0110

La tabla 1.3 ilustra 4 casos (pero no son los únicos casos) donde se observan las cuatro combinaciones posibles de bits de acarreo y préstamo.

Sistema “Exceso a la N”

El sistema de “exceso a la N” tiene una estrategia distinta a los anteriores. Sirve para representar números enteros con signo. Funciona determinando un número de bits n y estableciendo una nueva referencia para el 0 en algún número a partir del cual los números superiores serán positivos y los números inferiores serán negativos. El número negativo de mayor módulo siempre será 0, el mayor número positivo de mayor módulo siempre será $2^n - 1$, y N representará 0. Para representar un número $n_1$en este sistema simplemente se suma al número el exceso N y se representa en binario puro: $n_{1\,[EN} = (n_1 + N)_{[BP}$. Para obtener inversamente el número a partir de su representación, simplemente se calcula su valor como binario puro y luego se le resta el exceso.

Por ejemplo, un sistema en Exceso a la 2^{n-1} con 8 bits, el nuevo cero es $2^{8-1} = 2^7 = 128$y por lo tanto la numeración se verá como en la tabla 1.4.

Tabla 1.4. Ejemplo del rango de un sistema Exceso a la N = 128

Representación E128	Operación	Valor representado
1111 1111	255 - 128	127
1111 1110	254 - 128	126
1000 0101	133 - 128	5
1000 0000	128 - 128	0
0111 1011	123 - 128	-5
0000 0000	0 - 128	-128

En este sistema hay un sólo cero, N, y el rango nunca puede ser simétrico.

Decimal codificado en binario

El sistema decimal codificado en binario (BCD, por sus siglas en inglés) es un tanto distinto a los anteriores, pero intuitivo y simple. Se basa en representar cada dígito de un número decimal en una cantidad fija de bits, haciendo corresponder el valor binario puro de la secuencia de bits con el valor del dígito. Por supuesto, 3 bits no serían suficientes, porque permiten representar 8 valores, o, dicho de otra manera, sólo los números del 0 al 7. El sistema puede utilizarse con grupos a partir de 4 bits (recordemos que un conjunto de 4 bits se denomina *nibble*), y como 4 es el menor número en que entran todos los dígitos decimales, es una de las opciones más utilizadas. Este sistema se conoce como "BCD empaquetado" porque los nibbles se almacenan en conjuntos dentro de bytes o palabras más largas, y para recuperar cada dígito es necesario luego "desempaquetar" cada nibble. En contraste, cuando en lugar de 4 bits se selecciona una cantidad de bits mayor, usualmente un byte o el tamaño de palabra del procesador, se conoce como BCD "desempaquetado" porque el valor puede leerse directamente tomando el contenido completo de la palabra. La diferencia se explicita en el siguiente ejemplo donde se escribe el número $209_{(10}$ de dos maneras distintas:

$209_{(10}$

BCD empaquetado	BCD desempaquetado
0010 0000 1001	0000 0010 0000 0000 0000 1001

Cada uno de los dígitos de 209 se representa con 4 bits en BCD empaquetado: $2_{(10} = 0010_{(2}$, $0_{(10} = 0000_{(2}$y $9_{(10} = 1011_{(2}$. Luego esos nibbles se almacenan en una palabra. En cambio, para el BCD desempaquetado cada uno se coloca en una palabra (de 8 bits en el ejemplo). Por supuesto, de esta manera ocupan más espacio, pero son más fáciles de leer en un programa.

Existen variantes de este sistema que incluyen la posibilidad de representar números negativos, pero no se trata de un estándar tan común como los que hemos visto y cada caso debe estudiarse por separado.

Representación de números reales

Características de los números reales

Con los números enteros nos preocupaba el rango y la representación del cero. En el caso de los números reales, interesa además la *precisión* que alcanzan. Como vimos, los números enteros pueden representarse exactamente en binario, pero los números decimales en ocasiones no tienen una representación exacta porque, en base 2, resultan periódicos, irracionales, o cuando queramos hacerlos caber en un número fijo de bits, simplemente no alcanzarán los casilleros para representar el número completo.

Sistema "de punto fijo"

El primer sistema que podríamos imaginar para representar números reales es el de punto fijo (PF), en el cual se asigna una cantidad "fija" de bits a la parte entera del número, y otra cantidad fija a la parte fraccionaria. Además, se suele reservar un bit para el signo. Debido a que el punto fijo se implementa según la conveniencia de distintas aplicaciones no hay un único estándar difundido y la forma de representación varía. Sin embargo, el concepto que siempre se repite es la cantidad fija de bits para cada parte.

Usualmente el total de cifras binarias ocupa 8, 16, 32 o 64 bits. Por lo tanto, si se especifica un sistema de punto fijo con signo de 4 bits en un sistema de 8 bits, se sobreentiende que los 4 bits son para la parte fraccionaria, 1 será para el signo, y los restantes 3 son para la parte entera.

Por supuesto, cometeremos un cierto error al intentar representar números en este formato. Consideremos el número $+5{,}8125_{(10}$. Puede pasarse este valor a base 2 con los métodos vistos antes en este capítulo y se obtiene $+101{,}1101_{(2}$. Ese valor cabe exactamente en un sistema de representación de punto fijo de 8 bits con 4 asignados a la parte fraccionaria y 3 a la parte entera:

Valor: $+5{,}8125_{(10}$ Número binario: $+101{,}1101_{(2}$ PF 1-3-4: `0101 1101`

En cambio, consideremos ahora el valor +5.8126. Si realizamos la conversión de base se obtiene $+101.110101001010\ldots_{(2}$ y al representarlo en punto fijo con el formato 1-3-4 queda:

Valor: $+5{,}8126$ Número binario: $+101{,}110101001010\ldots_{(2}$ PF 1-3-4: `0101 1101`

Que rápidamente reconocemos como la representación de $+5{,}8125_{(10}$. Por lo tanto, se está cometiendo un error absoluto de 0,0001 o relativo porcentual de 0,0017%. El paso más pequeño que puede darse entre un número y otro es incrementar en 1 el bit menos significativo del código binario obteniendo $101{,}1101_{(2} + 0{,}0001_{(2} = 101{,}1110_{(2} = 5{,}875_{(10}$. Es notorio que no podremos diferenciar valores entre estos dos pasos con este sistema. Efectivamente, se está truncando la representación en base 2 en el 4to dígito fraccionario y como vimos esto acota el error, y la distancia entre pasos discretos, a menos de $2^{-4} = 0{,}0625$.

Otro sistema de punto fijo se basa en el sistema BCD visto anteriormente, donde se asigna una cantidad de nibbles fija (en el caso de BCD empaquetado) a la parte entera y otra fija a la parte fraccionaria. En este formato también puede establecerse un código especial con los valores libres (ya que 4 bits representan 16 valores de los que ocupamos sólo 10 representando directamente números decimales) para indicar dónde está la coma. Una ventaja importante de este método es que mientras alcance la cantidad de nibbles, se representa en forma exacta el número decimal, ya que no se convierte a base 2. Sistemas similares a este son por lo tanto utilizados frecuentemente en aplicaciones de contabilidad.

Sistema de punto flotante IEEE 754

Comenzaremos la descripción de los sistemas de punto flotante comprendiendo sus ventajas a partir de comparar estos números: 0,005 y 0,0000000000005. Son números distintos, y en un

sistema de representación de punto fijo de 8 bits el segundo probablemente no podría representarse. Sin embargo, notemos la siguiente manera particular de describirlos a ambos:

0,005	2 ceros luego de la coma y un 5
0,0000000000005	12 ceros luego de la coma y un 5

Verdaderamente, la información que describe ambos números es muy similar y de hecho esto se aprovecha en la notación exponencial habitual en matemática:

$$0{,}05 \; = \; 5 \; \times 10^{-2}$$

$$0{,}0000000000005 \; = \; 5 \; \times 10^{-12}$$

Donde se ve que la información importante es el número 5, la base 10, y el exponente (-2 y -12 en cada caso).

La notación en punto flotante es muy útil y la organización IEEE estableció un estándar para la representación en punto flotante que se hizo muy popular y se implementa hoy en día en prácticamente todos los lenguajes de programación.

El estándar propone expresar el número en notación exponencial en base 2 normalizada y luego almacenar el signo, exponente y mantisa. El método por lo tanto comienza por los siguientes pasos previos:

1. Pasar el número a base 2
2. Expresarlo en forma normalizada
3. Identificar el signo, el exponente y la mantisa

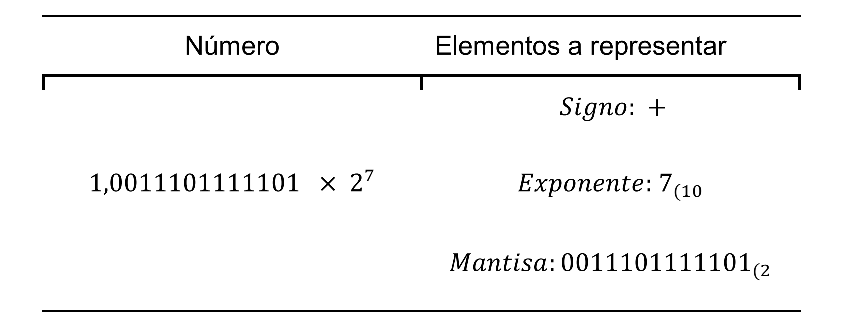
Ejemplificamos con el número $157{,}953125_{(10}$. Como primer paso se escribe en base 2:

$$157{,}953125_{(10} \; = \; 10011101{,}111101_{(2}.$$

El segundo paso, la normalización, consiste en expresar el número en notación exponencial de manera de dejar solamente un 1 en la parte entera:

$$10011101{,}111101 \; = 1{,}0011101111101 \; \times \; 2^{7}$$

Los dos números son equivalentes en valor, pero están escritos de manera distinta. Este paso se puede realizar rápidamente ya que multiplicar o dividir por 2 en base dos es “correr la coma”. Continuamos con el tercer paso identificando sus elementos significativos:



Número	Elementos a representar
$1{,}0011101111101 \; \times \; 2^{7}$	$Signo\colon +$ $Exponente\colon 7_{(10}$ $Mantisa\colon 0011101111101_{(2}$

Notemos que, si *siempre* se normaliza el número dejando un 1 delante, entonces no aporta información extra anotarlo, realmente no es necesario, y por lo tanto se adoptó la

forma de notación denominada “con bit oculto” por la cual se registra únicamente la mantisa y se omite ese bit.

Una vez que tenemos todos los elementos, el siguiente paso es conocer cómo se codifica cada uno formando el número completo. Existen tres opciones: el formato de simple precisión que ocupa 32 bits, el formato de doble precisión que ocupa 64 y el formato extendido que ocupa 128.

Para el formato de simple precisión

- El signo ocupa 1 bit, el más significativo y se codifica con un 1 si es negativo, y con un 0 si es positivo.
- El exponente ocupa los 8 bits siguientes y se codifica en formato Exceso a la $2^{N-1} - 1$, que en simple precisión con $N = 8$ resulta 127.
- La mantisa se representa en los restantes 23, directamente en binario alineada a la izquierda.

Con estas reglas el número del ejemplo $+1{,}0011101111101 \times 2^{7}$, resulta

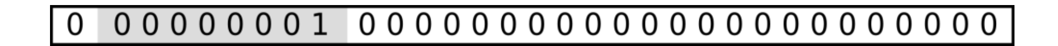
31 30 23 22 2 1 0
0 1 0 0 0 0 1 1 0 0 0 1 1 1 0 1 1 1 1 1 0 1 0 0 0 0 0 0 0 0 0 0
Signo Exponente Mantisa

Ya que el exponente se representa en Exceso a 127 en 8 bits, su rango sería de -127 a 128. Sin embargo, los códigos 00000000 (que correspondería al -127) y 11111111 (que correspondería a 128) tienen un significado especial como se muestra en la tabla 1.5.

En primer lugar, se reserva el código 1111 1111 y una mantisa igual a 0 para el concepto de infinito positivo y negativo, que puede suceder por *overflow* o porque se divide por cero, entre otras operaciones.

En segundo lugar, se reserva el código 1111 1111 y una mantisa distinta de 0 para representar el concepto de un valor que surgió por un estado de error o falta de inicialización y por lo tanto “no es un número” (*not a number* en inglés o NaN).

Finalmente, reserva el 0000 0000 y una mantisa distinta de 0 para un caso especial que soluciona un problema que surge por la normalización. Si examinamos con cuidado la normalización, vemos que al representar el valor más pequeño posible en formato normalizado tenemos:



Que es equivalente a $1{,}0000\ldots00 \times 2^{-126}$

Tabla 1.5. Rango completo de valores del sistema de representación IEEE754 simple precisión

Signo	Exponente	Mantisa	Valor representado
s	1111 1111	0	$(-1)^s \times \infty$
s	1111 1111	$\neq 0$	$(-1)^s \times$NaN
s	1111 1110	X	$(-1)^s \times (1+X) \times 2^{127}$
s	1111 1101	X	$(-1)^s \times (1+X) \times 2^{126}$
		...	
s	0000 0010	X	$(-1)^s \times (1+X) \times 2^{-125}$
s	0000 0001	X	$(-1)^s \times (1+X) \times 2^{-126}$
s	0000 0000	0	$(-1)^s \times 0$
s	0000 0000	$X \neq 0$	$(-1)^s \times (\mathbf{0}+X) \times 2^{-126}$

Pero notamos que habría lugar para expresar un número aún más pequeño si no estuviéramos obligados a mantener el “bit oculto”. Así, sin ese molesto bit oculto podríamos representar valores más pequeños como $0{,}1 \times 2^{-126}$, o aprovechando los 23 lugares de mantisa que sabemos tiene el formato, puede llegarse a 1×2^{-149}como valor más pequeño. Esa exactamente es la previsión que los creadores del estándar tuvieron y por eso el exponente 00000000 en lugar de representar el exponente -127 se reserva para representar números sin bit oculto con exponente -126.

Los casos de doble precisión y precisión extendida son equivalentes, pero adaptando el esquema de 32 bits divididos en el formato 1:8:23, a 64 bits en doble precisión con el formato 1:11:52 y a 128 bits en precisión extendida con el formato 1:15:112.

Representación de caracteres

Hasta ahora hablamos de representar números en la computadora, un proceso con algunas complicaciones pero que resulta intuitivo porque cada número se codifica en base 2 con alguna transformación matemática. Cuando el tipo de información a representar no es numérico, la representación no es tan directa como una ecuación matemática. Después de los números la forma más fundamental de información que deseamos manejar en la computadora es el texto (y luego probablemente los memes). Sin embargo, sabemos que la computadora sólo puede representar números en su interior ¿cómo nos muestra texto en la pantalla?

Como en todos los sistemas de representación, para poder representar texto se establece una convención. Se propone que a cada letra del texto le corresponda un número específico, siempre el mismo, y de esa manera un periférico de entrada como el teclado, al presionar por ejemplo la letra ‘a’ cuenta con circuitos que envían el código que le corresponde a la letra ‘a’ a la

computadora, que lo almacena en su memoria. De la misma manera, para mostrar el carácter en pantalla, la computadora envía el código y una serie de programas y circuitos se encargan de mostrar en la pantalla la imagen que corresponde al carácter ‘a’.

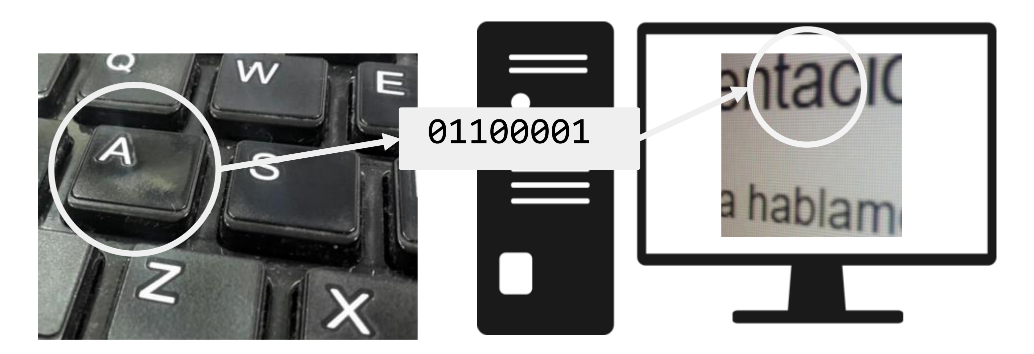


Figura 1.6. *Representación de la codificación del carácter ‘a’.*

Aquí conviene resaltar que todo lo que vemos como texto en la pantalla está constituido por caracteres, incluyendo también los números. Cuando la computadora nos muestra en pantalla un ‘1’, lo que nos está mostrando es en realidad el carácter que simboliza al número uno, un simple dibujo. Esto es importante porque ambas cosas no se almacenan de la misma manera en memoria: si almacenamos el número 1 en complemento a 2 de 8 bits tendremos 00000001 en un registro. Si almacenamos el carácter ‘1’ con alguna de las codificaciones que veremos a continuación, podríamos tener algo como esto: 00110001. Para diferenciar un valor numérico de un carácter utilizaremos la notación de comillas simples y diremos que ‘1’ es el carácter que representa al número 1.

Para continuar, es conveniente adoptar una perspectiva histórica y remontarse a la primera convención de caracteres ampliamente difundida que fue el estándar ASCII[10]. Quienes lo crearon eran programadores estadounidenses y en inglés no necesitaban caracteres como la ñ o los acentos, por lo que se enfocaron en un conjunto reducido de letras, números y símbolos como ‘!’, ‘?’, ’=’, entre otros.

También se incorporaron caracteres “no imprimibles”, que son códigos útiles para la comunicación pero que describen una acción en lugar de un símbolo que pueda dibujarse en pantalla, como la acción de retroceder un espacio, borrar una letra, avanzar una línea en el texto, entre otros.

Se creó así en 1963 una tabla de 128 valores para todos estos caracteres, lo cual se puede representar con 7 bits logrando que entre cómodamente en un byte. Esta tabla se difundió poco a poco en todo el mundo, sobre todo al ser adoptada por la primera computadora personal en 1981. En la figura 1.6 se representan algunos valores de esta tabla. Se pueden probar fácilmente

[10] ASCII es la sigla de *American Standard Code for Information Interchange* (Código Estadounidense Estándar para el Intercambio de Información). La motivación fue la necesidad de intercambiar texto entre dos computadoras. En verdad, sólo se intercambian números, se necesita estar de acuerdo en el estándar de representación para que ambas muestren el mismo texto.

los valores en cualquier editor de texto presionando la tecla ALT y escribiendo con el *pad* numérico del teclado el código ASCII correspondiente. En prácticamente todos los programas que haremos en este libro se presume el estándar ASCII para representar texto.

Más adelante, se crea ASCII extendido, con códigos entre 0 y 255; los primeros 128 valores son iguales en todo el mundo por provenir del ASCII original, pero no así los siguientes.

Tabla 1.6. Resumen demostrativo de la Tabla ASCII.

Rango	Descripción	Carácter
0	Carácter nulo	`NULL`
1-31	no imprimibles	
32	Espacio	‘ ’
48-57	Números	‘0’ - ‘9’
65-90	Letras mayúsculas	‘A’ - ‘Z’
97-122	Letras minúsculas	‘a’ - ‘z’

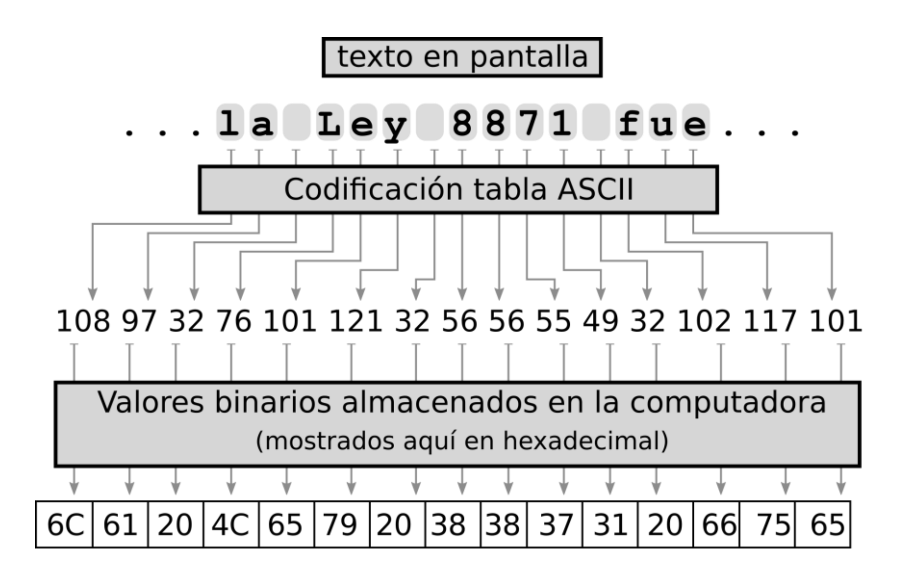


Figura 1.7. *Representación de texto en la computadora según tabla ASCII*

Hoy en día, con el auge de la comunicación basada en texto por e-mail, páginas de internet y aplicaciones de mensajería, se utilizan centenares de símbolos aparte de los presentes en la tabla ASCII extendida, como los emoticones, además de tenerse un contacto mucho más asiduo con lenguas con alfabetos distintos como el ruso, chino, japonés, etc. Interesó por tanto contar con un estándar universal que incorporara todos estos símbolos y fuese muy amplio y flexible.

Notemos que el ASCII establecía al mismo tiempo qué número asignar a cada letra y cómo representar ese número en la memoria de la computadora, es decir, cómo *codificar* ese valor. De esta manera, al carácter ‘a’ le corresponde el número de la tabla ASCII 97 y se guarda en la memoria de la computadora según la conversión $97_{(10} = 01100001_{(2}$.

Es muy difícil para un proyecto tan ambicioso como estandarizar todos los símbolos existentes alcanzar un acuerdo no solo en el orden e identificación de todos esos caracteres sino también en cómo almacenarlos en la computadora. Por lo tanto, la solución que se encontró fue establecer un estándar donde sólo se establece qué número o identificación le corresponde a cada carácter, y se llamó UNICODE. En UNICODE cada símbolo tiene un código asignado y se suele escribir como "U+" seguido del número en hexadecimal. Así, el *emoticón* clásico de la carita feliz es U+1F600 😀. Pero esto nada dice de cómo se debe almacenar el código U+1F600 en la computadora. El estándar más común de codificación es el UTF-8 que tiene la ventaja de tener una longitud variable entre 1 y 4 bytes: no sólo permite ahorrar espacio, sino que al representar los caracteres de la tabla ASCII con un solo byte es totalmente compatible con textos más antiguos en codificación ASCII.

Ejercicios

1) Realice las siguientes conversiones de base
   a) $10011010{,}01_{(2}$ a $B_{(10}$
   b) $8EAD_{(16}$ a $B_{(2}$
   c) $10257{,}4_{(8}$ a $B_{(16}$
   d) $123{,}04_{(5}$ a $B_{(2}$
   e) $0.48_{(10}$ a $B_{(6}$

2) Escriba:
   a) Las tablas de conversión entre base 2, 8 y 16, con sus valores en base 10 en una cuarta columna.
   b) El número 1AF32 en base 2 utilizando la conversión directa por tabla.

3) Realice las siguientes operaciones en la base indicada:
   a) $9_{(10} + 1_{(10}$
   b) $1_{(2} + 1_{(2}$
   c) $7_{(8} + 1_{(8}$
   d) $F_{(16} + 1_{(16}$

4) Efectúe las siguientes operaciones en la base indicada.
   a) $10101100_{(2} + 00001111_{(2}$
   b) $11011101_{(2} * 1000_{(2}$
   c) $A2356_{(16} + C2F3_{(16}$
   d) $B1206_{(16} * A1_{(16}$

5) Indique en que bases son ciertas las siguientes afirmaciones (note que si los números estuviesen en base 10, estas ecuaciones no se cumplirían, B es distinta para los distintos incisos)
   a) $3_{(B} * 14_{(B} – 50_{(B} = 0_{(B}$
   b) $123_{(B} * 32_{(B} = 10041_{(B}$
   Nota: Quizás le sea útil utilizar el teorema fundamental de la numeración.

6) Realice un programa en C para convertir un número binario de largo arbitrario (menor a 100 cifras) ingresado como cadena de carácter a hexadecimal y a base 10. El programa debe mostrar, primero, al número separado en Bytes. Debajo de cada byte debe mostrar su valor hexadecimal correspondiente, y finalmente imprimir el valor base 10.

7) Represente los números enteros $17_{(10}$ y $–5_{(10}$ en las siguientes formas. (Considere que el ordenador trabaja con palabras de 8 bits).
   a) Módulo y Signo
   b) Complemento a 1
   c) Complemento a 2
   d) Exceso a 2 N-1

8) ¿Cuántos bits se necesitan como mínimo para representar los números 1023, 1024, y 50000 en binario puro?

9) ¿Qué valor decimal se obtiene al efectuar la siguiente operación?
   $01000000_{(ca2} + 11111110_{(ca2}$

10) ¿Qué valor está antes y después del siguiente valor en Ca2?
   $10000001_{(ca2}$

11) Escriba el número 100,062255859375 en coma fija de 16 bits y verifique en cada caso el error cometido para los siguientes formatos de coma fija:
   a) 1 bit de signo, 7 para la parte entera, y 8 para la parte fraccionaria
   b) 1 bit de signo, 13 para la parte entera, y 2 para la fraccionaria
   c) 1 bit de signo, 2 para la parte entera, y 13 para la parte fraccionaria.

12) ¿Cuál es la cantidad de bits que necesita para representar la parte fraccionaria de un sistema de coma fija con error menor a 2^{-10}?

13) Represente en coma flotante simple precisión:
   a) $29_{(10}$
   b) $-0.163_{(10}$
   c) cero, NAN,-NAN, Inf, -Inf

14) La memoria de la computadora contiene la siguiente secuencia de bits:

    0101 0101   0100 1110   0100 1100   0101 0000

    Encuentre la información que representa si se trata de
    a) Una secuencia de 4 caracteres ASCII.
    b) Una secuencia de números enteros representados en MS de 8 bits.
    c) Un número en coma flotante simple precisión.

    Finalmente, exprese la secuencia de números en base 16.

15) Comente brevemente los problemas de representación en los sistemas de representación para:
    a) Enteros
    b) Reales coma fija
    c) Reales coma flotante

16) La función “printf” necesita conocer el sistema de representación del valor dado para imprimirlo correctamente. %d implica que el sistema es complemento a 2. %c se utiliza para caracteres ASCII. Agregue ahora las siguientes líneas e interprete la salida:

```
printf("Un valor: %d\n",90);
printf("Un valor: %c\n",90);
printf("Un valor: %d\n",0xFFFFFFFE);
printf("Un valor: %u\n",0xFFFFFFFE);
```

CAPÍTULO 2
Fundamentos básicos de programación y algoritmos

Marcelo A. Haberman

En este capítulo se introduce al lector en los aspectos más básicos de la programación, como fase previa al aprendizaje del lenguaje C.

En la primera parte se explica el esquema tradicional (en cascada) para la resolución de un problema mediante un programa informático. Se expone el concepto de algoritmo, sus elementos básicos (datos de E/S, variables, instrucciones ejecutables) y distintas maneras de representación de alto nivel (diagrama de flujo, pseudo-código).

En una segunda parte se desarrollan las distintas estructuras de control de flujo como bloques elementales del paradigma de "programación estructurada" y el concepto de función como herramienta fundamental para la "programación modular".

¿Qué es programar?

Programar es, en un sentido general, planificar y ordenar las acciones necesarias para realizar un proyecto. En nuestro caso, que nos ocuparemos de la programación de computadoras, programar es una actividad (para algunos es un arte) mediante la cual le indicamos la *secuencia de acciones* que debe realizar una máquina para cumplir con algún objetivo determinado.

Pero aunque parezca que programar una computadora es simplemente escribir las *instrucciones* que queremos que esta cumpla, la resolución de problemas mediante la creación de *programas informáticos* (*software*) suele ser un *proceso* arduo que comienza siempre con una necesidad o problema a resolver y, al contrario de lo que se cree, no finaliza con la obtención de un *programa ejecutable*, sino que el mismo estará sujeto a pruebas, modificaciones y actualizaciones.

Procesos de desarrollo del software

El proceso de desarrollo o "ciclo de vida" del software es una de las ramas de estudio de la Ingeniería de Software. Existen distintos modelos que describen el proceso de desarrollo, los

cuales plantean distintas etapas y formas de proceder para que un programador o un equipo de programadores lleve a cabo el desarrollo de un programa.

Modelo en cascada

El modelo de *desarrollo en cascada* (ver figura 2.1) es el más clásico, el primero en aparecer y a partir del cual surgen las técnicas más modernas.

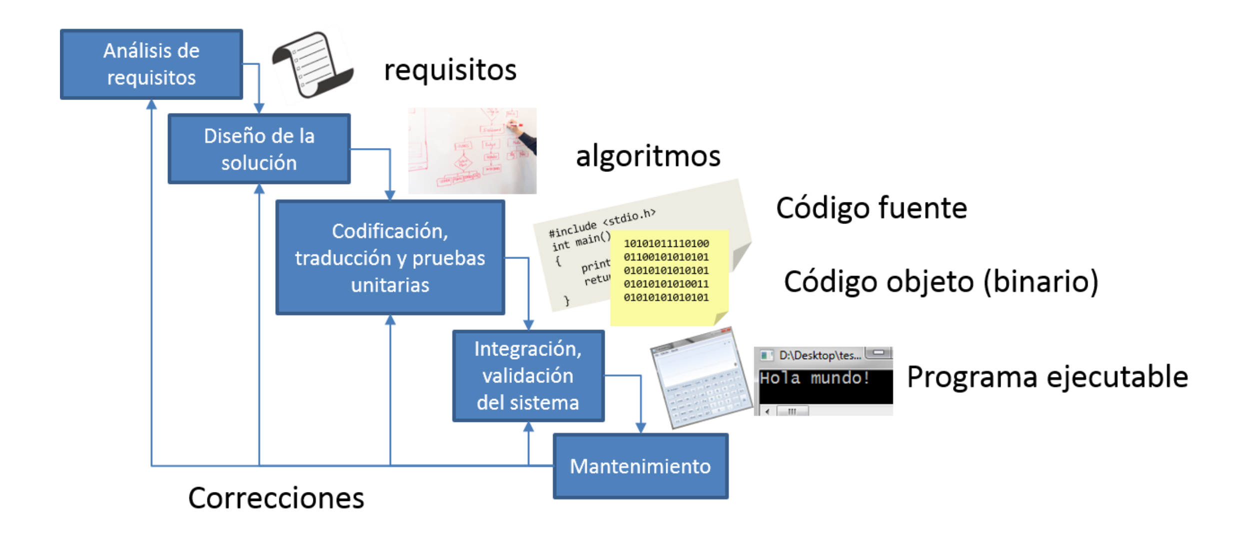


Figura 2.1: *Modelo de desarrollo en cascada.*

En este modelo se plantea seguir una serie de etapas de manera secuencial, las cuales resumimos como:

Análisis de requisitos

Se analiza la problemática a resolver y las necesidades de los futuros usuarios del programa, convirtiéndolos en *requisitos* concretos que debe cumplir el programa.

Diseño de la solución

Se realizan los esquemas de las distintas partes del software (*módulos, funciones*). Se determina la manera en que se representará y manejará la información de interés (*variables, constantes, estructuras de datos*). Se establece la forma de interacción del usuario (*interfaz, entradas y salidas*). Se diseñan los *algoritmos* de cada módulo.

En lo que resta de este capítulo se abordan los fundamentos para poder llevar a cabo la fase de diseño.

Codificación, traducción y pruebas unitarias

Se escribe el *código fuente* de los algoritmos de los distintos módulos en algún *lenguaje de programación*.

Se “traduce” el código escrito en un *lenguaje de programación* (comprensible y cercano a las personas) al *código de máquina* compuesto por el *conjunto de instrucciones* del procesador de la computadora, que sólo comprende y puede ejecutar este último. Existen dos tipos de “traductores”: *compiladores* e *intérpretes*, los cuales se explicarán en el siguiente capítulo.

Luego de la compilación (o interpretación) se prueba cada módulo de manera independiente del resto, verificando las salidas obtenidas para entradas conocidas.

Integración, validación y verificación del programa

Se integran los distintos módulos ya depurados, conformando el *programa ejecutable*. Se ejecuta el programa y se verifica el cumplimiento de los distintos requisitos generados en la fase de análisis. Se corrigen los errores encontrados.

Finalmente, se instala o despliega el programa en el sistema informático del usuario, que debe comprobar que el programa sea acorde con sus necesidades originales.

Mantenimiento

Corrección de fallas detectadas durante el uso y de necesidades cubiertas parcialmente.

Comentarios sobre la aplicación del modelo en cascada

Seguir a rajatabla el modelo anterior no es eficiente, especialmente cuando esperamos concluir toda la fase de codificación para comenzar la traducción y las pruebas. Lo cual retrasa la identificación de errores de funcionamiento y dificulta su identificación en el código.

Suele ser conveniente entremezclar estas dos fases, de manera tal de ir codificando, ejecutando y probando de a partes pequeñas de código. Así es posible identificar con más facilidad las secciones de código con errores y solucionarlos a la brevedad.

Algoritmos

Según el diccionario de la RAE un algoritmo es un “*Conjunto ordenado y finito de operaciones que permite hallar la solución de un problema.*”. Prácticamente convivimos con distintos algoritmos toda la vida, que permiten resolver desde los más triviales problemas de la vida cotidiana, como la secuencia de pasos para atarnos los cordones o las instrucciones para seguir una receta de cocina hasta procedimientos rutinarios de trabajo o protocolos de evacuación de un edificio ante una contingencia.

Como hemos visto en la sección anterior, para la informática, programar implica definir las acciones que deberá ejecutar una máquina (y el orden de las mismas) para cumplir un objetivo. Esto no es ni más ni menos que “decirle” a la computadora que debe ejecutar un algoritmo.

En general se deben cumplir las siguientes condiciones:

- En el algoritmo debe existir un único punto de *inicio* y (al menos) un fin.
- El algoritmo debe tener una *cantidad finita* de pasos.
- Los pasos del algoritmo deben ejecutarse en *tiempo finito*.
- Las órdenes deben ser *ejecutables*: Tienen que ser operaciones básicas que la computadora pueda realizar. También pueden ser procedimientos complejos, pero definidos en base a operaciones básicas y otros procedimientos, que puedan ser ejecutados.
- Las instrucciones deben ser *precisas*, no pueden ser ambiguas: Una orden o instrucción dada a una computadora solo puede ser interpretada de una única manera.

En el proceso de desarrollo de software, el diseño de los algoritmos es una parte fundamental ya que establece la secuencia de pasos lógicos, acciones y decisiones que posteriormente deberá ejecutar la computadora. Los algoritmos diseñados son independientes de los distintos *lenguajes de programación* y de las *arquitecturas de cómputo*, y constituyen descripciones de alto nivel de abstracción, que en una etapa posterior pueden codificarse en el lenguaje deseado. Esto permite trabajar sobre lo que debe hacer un programa sin que importen los detalles de cómo está codificado ni con qué lenguaje, yendo de lo más general a lo más particular[11].

Fundamentos y definiciones

Antes de profundizar en el diseño y representación de algoritmos existen tres conceptos fundamentales con los que el lector deberá familiarizarse, estos son los *Datos*, los *Operadores* y las *Funciones de E/S*.

Datos

Los conceptos vistos sobre los algoritmos hasta el momento giran en torno a las instrucciones o pasos que se deben ejecutar para cumplir un determinado objetivo. Sin embargo, la figura principal en el mundo de los algoritmos informáticos es la *información*, o los *datos* a través de los cuales esta se materializa.

Entradas y salidas

Suele ser común entonces pensar los algoritmos como bloques que procesan información, que tienen como materia prima un conjunto de *datos de entrada*, los cuales son manipulados por el algoritmo, y a partir de los cuales entrega otro conjunto de *datos de salida*. La figura 2.2 representa gráficamente este modelo.

[11] A esta modalidad de diseño desde lo más general (y abstracto) a lo más específico (y concreto) se lo conoce como "top-down", como se explica en el Capítulo 1.

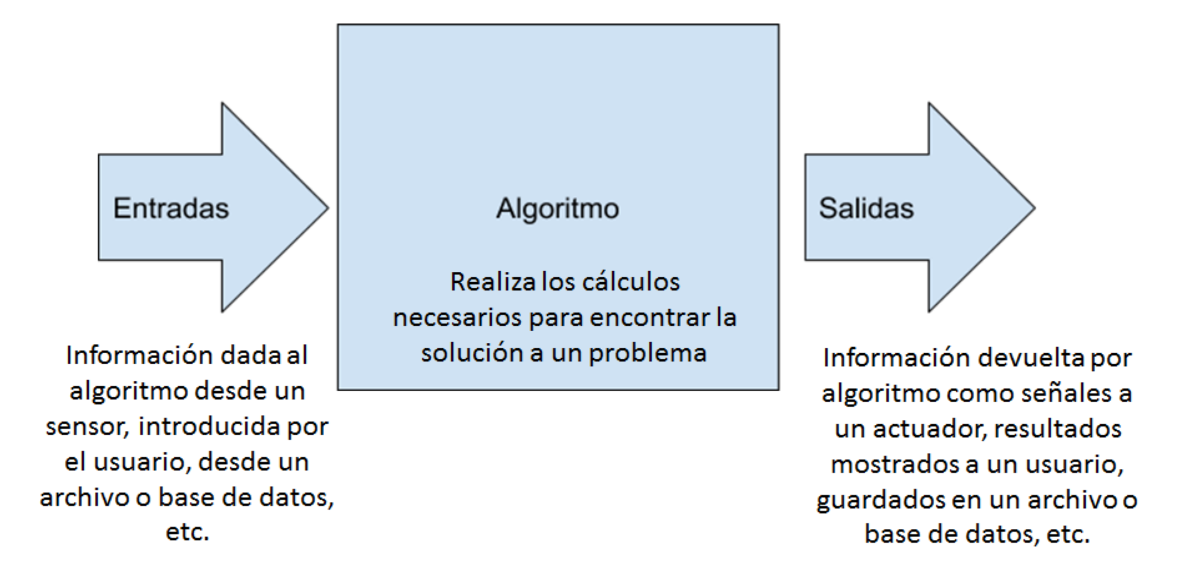


Figura 2.2: *Modelo del algoritmo como procesador de información*

Cuando se requiere diseñar o analizar un algoritmo es importante identificar cuáles son los datos de entrada y de salida.

Los datos de entrada pueden provenir de diversas fuentes, pueden ser valores previamente almacenados en la memoria principal o cualquier dato proveniente de los distintos periféricos de entrada del sistema de cómputo (como por ejemplo teclado, mouse, red, disco, scanner, sensor de temperatura, sensor de velocidad, etc.). En este curso trataremos principalmente con datos que provienen de tres fuentes distintas: datos ingresados por el usuario a través del teclado, parámetros de entrada de una función o programa y archivos en disco.

Los datos de salida son los que el algoritmo envía a un periférico de salida del sistema de cómputo (por ejemplo monitor, impresora, parlantes, red, disco, motores, luces, etc.); y también cualquier dato producido por el algoritmo que luego de finalizado el mismo queda en memoria principal a disposición de otros programas.

Variables

Tanto los datos de entrada/salida como el resto de los datos auxiliares que requiera un algoritmo para cumplir su objetivo, se almacenan en la memoria principal de la computadora, cada uno en un lugar (*dirección de memoria*) particular de la misma y ocupando un bloque de una cantidad determinada de esta memoria (medida en *bytes*). A cada uno de estos bloques de memoria donde se guarda un dato se lo conoce como *variable*. En el diseño del algoritmo, en principio no debemos preocuparnos por aspectos tales como las direcciones de memoria, la cantidad de bytes que ocupa cada dato, o su representación en memoria, sino que basta con referirnos a la variable con un nombre o *identificador*.

Por ejemplo, en la figura 2.3 se esquematiza como dos datos distintos, que son almacenados en posiciones arbitrarias de la memoria principal, abarcando bloques de varios bytes cada una, para el algoritmo son representadas por las variables reales *x* y *r*.

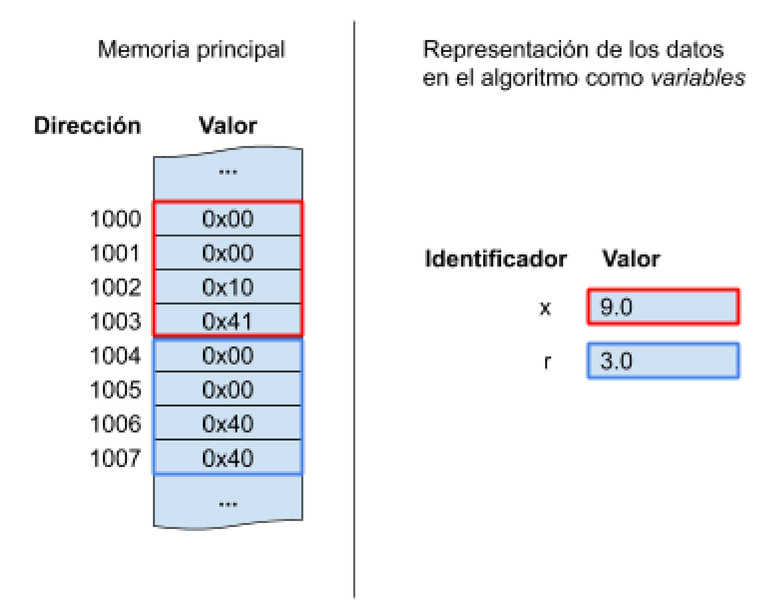


Figura 2.3. Representación en memoria principal de las variables reales *x* y *r* utilizadas por el algoritmo.

Tipos de datos

Las computadoras son esencialmente máquinas que realizan operaciones aritméticas entre números y operaciones lógicas entre valores lógicos (verdadero/falso), también llamados *booleanos*. Sin embargo, la naturaleza de la información que la computadora debe procesar puede ser de diversos tipos. Para diseñar un algoritmo de manera independiente del lenguaje de programación que se use posteriormente para su codificación, puede considerarse que prácticamente todos los lenguajes permitirán representar datos de tipo numérico, texto y variables lógicas.

Numéricos

Muchas arquitecturas básicas de hardware sólo pueden operar con *números naturales* y *números enteros* siendo la representación y la operación con *números reales* muy costosa computacionalmente, ya que se resuelve por software (mediante algoritmos específicos). Por lo tanto (casi)[12] todos los lenguajes de programación diferencian los tipos de datos numéricos entre reales y enteros, y para estos últimos suelen diferenciar entre enteros con signo y sin signo (naturales).

Lógicos

La gran mayoría de lenguajes de programación poseen un tipo de dato específico para las variables booleanas y los que no, como el lenguaje C, utilizan números enteros y convenciones como considerar “falso” al cero y “verdadero” a todo lo que no sea cero.

Texto

Más allá de trabajar solamente con datos lógicos y numéricos, las computadoras que son operadas por el ser humano deben interpretar las entradas de los usuarios y reportar resultados

[12] Una excepción notable es el lenguaje JavaScript que tiene un tipo de dato único para representar números que, por supuesto, permite almacenar reales.

legibles por estos últimos. Por esto prácticamente cualquier lenguaje de programación posee el tipo *carácter* para representar letras, símbolos alfanuméricos y de puntuación. Agrupando varios caracteres es posible representar texto. Muchos lenguajes, además, poseen un tipo de dato específico llamado *cadena* para el almacenamiento de texto.

Otros tipos especiales

Es común que los lenguajes incluyan algún tipo de dato compuesto para representar *fechas*. Los programadores, a su vez, pueden crear sus propios *tipos de datos compuestos*, agrupando variables de distintos tipos en una única *estructura*. Este tema se profundizará en el capítulo 7**.**

Arreglos

Los arreglos son conjuntos de variables del mismo tipo que comparten un mismo identificador, y donde cada uno de los elementos que componen el arreglo se identifica por un índice. Este tema se profundizará en el capítulo 4.

Constantes

Las constantes son tratadas como variables cuyo valor no podrá modificarse durante la ejecución del programa.

Operadores

Son elementos de los lenguajes de programación que definen las operaciones más básicas que pueden realizarse sobre los datos (operandos). La combinación entre operadores y operandos forman *expresiones*, las cuales dan como resultado un **valor** de un **tipo** determinado.

Operador de asignación

Modifican el valor almacenado en una variable. El identificador de la variable se escribe a la izquierda del operador y a la derecha va la expresión a evaluar, cuyo resultado se almacenará en la variable mencionada.

Para su representación se usarán indistintamente cualquiera de dos símbolos: una flecha de derecha a izquierda, que indica el sentido de la asignación: **<-**, o el operador igual: **=** .

Tabla 2.1: operadores de asignación
(los operadores en rojo no son válidos en lenguaje C)

Operador	Ejemplos
<-	a <- 5.6 b <- raiz_cuadrada(a*10)
=	a = 5.6 b = raiz_cuadrada(a*10)

Operadores aritméticos

Representan las operaciones matemáticas más comunes entre números. Los operandos y el resultado de la operación son numéricos.

Tabla 2.2: operadores aritméticos
(los operadores en rojo no son válidos en lenguaje C)

Operador	Operación	Tipo operandos	Tipo resultado	Ejemplos	Resultado
+	suma	numéricos	numérico	5 + 14	19
-	resta	numéricos	numérico	5 - 14	-9
*	multiplicación	numéricos	numérico	5 * 14	70
/	división	numéricos	real	5 / 14 14 / 5	0.3571 2.8
mod (%)	módulo (resto de división entera)	enteros	entero	5 mod 14 14 mod 5	5 4
div	división entera	enteros	entero	5 div 14 14 div 5	0 2
^	potencia	numéricos	numérico	14 ^ 5	537824

Operadores relacionales

Comparan dos operandos numéricos y devuelven un resultado lógico.

Tabla 2.3: operadores relacionales
(los operadores en rojo no son válidos en lenguaje C)

Operador	Operación	Tipo operandos	Tipo resultado	Ejemplos	Resultado V: verdadero F: falso
==	igual a	numéricos	lógico	5 == 14	F
>	mayor que	numéricos	lógico	5 > 14	F
<	menor que	numéricos	lógico	5 < 14	V
>=	mayor o igual que	numéricos	lógico	5 >= 14	F
<=	menor o igual que	numéricos	lógico	5 <= 14	V
<> (!=)	distinto a	numéricos	lógico	5 <> 14	V

Operadores lógicos

Realizan las 3 funciones lógicas básicas (NO,Y,O) cuya *tabla de verdad* se presenta en la Tabla 2.4.

Tabla 2.4: tabla de verdad de las operaciones lógicas

A	B	no(A)	no(B)	A y B	A o B
F	F	V	V	F	F
F	V	V	F	F	V
V	F	F	V	F	V
V	V	F	F	V	V

Por su parte, el uso de los operadores lógicos se resume en la siguiente tabla:

Tabla 2.5: operadores lógicos
(los operadores en rojo no son válidos en lenguaje C)

Operador	Operación	Tipo operandos	Tipo resultado	Ejemplos	Resultado
NO **(!)**	negación: opera sobre un único operando, invirtiendo el valor lógico	lógico	lógico	no(5 < 14) no(5 == 14)	F V
Y / AND **(&&)**	conjunción: el resultado solo es verdadero si ambos operandos son verdaderos, de lo contrario es falso.	lógicos	lógico	no(5 < 14) y no(5 == 14) 5 < 14 y no(5 == 14)	F V
O / OR **(||)**	disyunción: el resultado es verdadero si al menos uno de los operandos son verdaderos, de lo contrario es falso.	lógicos	lógico	no(5 < 14) o no(5 == 14) 5 < 14 o no(5 == 14) no(5 < 14) o (5 == 14)	V V F

Precedencia

La precedencia o prioridad de los operadores determina el orden en que se van a evaluar las distintas operaciones en una expresión compleja en la que intervienen varios operadores y sus respectivos operandos. Los operadores de mayor prioridad se evaluarán antes que los de prioridades más bajas. Los paréntesis sirven para delimitar expresiones y tienen la prioridad más alta. En caso de haber varios operadores de un mismo nivel de prioridad, estos se evalúan según la asociatividad de cada nivel, de izquierda a derecha en general o de derecha a izquierda para la asignación.

Tabla 2.6: prioridad de evaluación para los operadores de una expresión algorítmica. Esto en general se cumple para todos los lenguajes.

Operador	Prioridad	Asociatividad
()	Más alta (se evalúa primero)	izquierda a derecha
^		izquierda a derecha
* / mod div		izquierda a derecha
+ -		izquierda a derecha
==, <>, <, >, <=, >=		izquierda a derecha
NO		izquierda a derecha
Y		izquierda a derecha
O		izquierda a derecha
=	Más baja (se evalúa último)	derecha a izquierda

Ejemplo 2.1:

Analizar el orden en que se evalúan las operaciones en cada expresión, y los valores que quedarán almacenados en las variables `a`, `b`, `c` y `d`, y el tipo de dato de la variable. Nótese que en un algoritmo las instrucciones (*sentencias*) se ejecutarán en orden, y en este caso los números a la izquierda de cada línea indican dicho orden de ejecución.

```
1   a = 4,5 * 4 div 1 + 3
2   b = 4,5 * 4 div (1 + 3)
3   c = 4,5 * (4 div (1 + 3))
4   d = a <> b Y a<>c Y NO(b==c)
```

Resolución:

Resolveremos cada línea por separado, evaluando las operaciones según el orden de prioridad dado por la tabla 2.6.

Linea 1:

`a = 4,5 * 4 div 1 + 3` se indican en rojo los operadores involucrados

`a = 4,5 * 4 div 1 + 3` en azul los operadores de mayor prioridad

`a = 4,5 * 4 div 1 + 3` como hay más de uno se evalúa de izquierda a derecha

`a =    18   div 1 + 3`

`a =         18    + 3`

`a =                21` → Se determina el valor de la variable `a`, que puede ser **entera o real**.

Linea 2:

```
b = 4,5 * 4 div (1 + 3)
b = 4,5 * 4 div (1 + 3)      Lo que esté encerrado entre paréntesis se evaluará primero
b = 4,5 * 4 div ( 4 )
b = 4,5 * 4 div   4
b =    18   div   4
b =           4              Se determina el valor de la variable b, que puede se entera o real.
```

Linea 3:

```
c = 4,5 * (4 div (1 + 3))
c = 4,5 * (4 div (1 + 3))    Lo que esté encerrado entre paréntesis se evaluará primero
c = 4,5 * (4 div (1 + 3))    Si hay paréntesis anidados se evaluarán primero
c = 4,5 * (4 div (  4  ))
c = 4,5 * (4 div    4)
c = 4,5 *       1
c =      4,5                 Se determina el valor de la variable c, que será real.
```

Linea 4:

```
d = a <> b Y a<>c Y NO(b==c)    Cuando se ejecuta la líne 4: a= 21, b= 4 y c= 4,5
d = a <> b Y a<>c Y NO(b==c)
d = a <> b Y a<>c Y NO(F)
d = a <> b Y a<>c Y NO F
d = a <> b Y a<>c Y NO F
d =      V     Y a<>c Y NO F
d =      V     Y    V    Y NO F
d =      V     Y    V    Y  V
d =            V         Y  V
d =                      V      Se determina el valor de la variable d, que será lógica.
```

Funciones predefinidas

Además de los operadores mencionados, que permiten expresar operaciones aritmético-lógicas elementales, en general los programadores contarán con un conjunto de funciones predefinidas para la plataforma en la cual desarrollan. Estas funciones se identifican mediante un nombre particular (al igual que las variables y constantes) y pueden ser *llamadas* o *invocadas* por el programador desde un algoritmo principal.

En lugar de realizar operaciones simples, como las que realizan los operadores, pueden requerir la ejecución de numerosos pasos para realizar una tarea concreta, es decir que son sub-algoritmos o sub-programas, *invocados* desde otro algoritmo. Las funciones, además, pueden recibir *parámetros* como datos de entrada desde el algoritmo que las invoca y devolver a este algún resultado como salida. Si bien estas funciones con que se cuenta dependen tanto de las herramientas de desarrollo, como del lenguaje y del hardware utilizados, en general se dispone de *bibliotecas* (conjuntos de funciones) para las siguientes categorías:

- Entrada / salida: Las funciones de entrada brindan los mecanismos para que un algoritmo pueda recibir información desde los periféricos de entrada (teclado, ratón, micrófono, cámara, red, sistema de archivos, sensores, etc.). Por su parte, las funciones de salida proporcionan al algoritmo una manera de enviar la información procesada a los periféricos de salida (pantalla, impresora, parlantes, red, sistema de archivos, actuadores, etc.). Para algoritmos que interactúan con usuarios a través de *consolas de texto*, la entrada típica de información al algoritmo es el ingreso de texto a través del teclado y la salida de información del algoritmo al usuario es mediante texto escrito en la pantalla. Para trabajar con estas entradas y salidas generalmente existen las funciones `Leer()` y `Escribir()` o equivalentes.
- Matemáticas: Permiten realizar operaciones matemáticas más complejas, como logaritmos, funciones trigonométricas, exponenciales, raíces, redondeos, etc.
- Manejo de texto: Facilitan el procesamiento de textos. Permiten unir y separar cadenas, pasar a mayúsculas o minúsculas, contar letras, comparar textos, etc.
- Manejo de fechas / horas: Asisten en el procesamiento de datos de fecha/hora.

Ejemplo 2.2:

(a) Escribir las instrucciones que formarán parte de un algoritmo que calcula el perímetro y área de una circunferencia. El algoritmo debe interactuar con un usuario que ingresará el radio por teclado. Los resultados se mostrarán en pantalla.

(b) Mencione las variables que intervienen y si almacenan datos de entrada o de salida.

(c) Simular la ejecución instrucción por instrucción y el estado de las variables suponiendo que el usuario ingresa un radio de 1.5 m.

Resolución:

(a) El programa deberá esperar a que el usuario ingrese un valor numérico por teclado para el radio de la circunferencia (en metros), para eso se usará la función `Leer()`, que recibe como parámetro la variable donde se almacenará el valor ingresado:

```
Leer(radio)
```

Luego, se pueden calcular el área y el perímetro, los cuales se guardarán en dos variables distintas

```
Leer(radio)
área = radio*radio*3.1416
perímetro = 2*radio*3.1416
```

Finalmente, el algoritmo debe mostrar al usuario los resultados de su procesamiento, para esto usamos la función `Escribir()`, que recibe como parámetro la información a mostrar, según las siguientes convenciones:

- El texto literal se pasa entre comillas dobles: `Escribir("texto")`
- Para mostrar el valor de una variable se pasa su identificador sin comillas: `Escribir(variable)`

- Para escribir en la siguiente línea se pasa el parámetro `NL` (nueva línea): `Escribir(NL)`
- Pueden pasarse varios parámetros separados por comas, se escribirán uno a continuación del otro: `Escribir("valor 1 = ", var1, " valor 2 = ", var2)`

```
1    Leer(radio)
2    área = radio*radio*3.1416
3    perímetro = 2*radio*3.1416
4    Escribir("El área es ", área, " m2",NL)
5    Escribir("El perímetro es ", perímetro, " m")
```

Nótese que al comenzar el programa el usuario no recibe ninguna indicación del programa, lo cual puede ser confuso. Para corregir esto, se agrega una instrucción al principio para mostrar un cartel con indicaciones al usuario.

```
0    Escribir("Ingrese el radio en metros: ")
1    Leer(radio)
2    área = radio*radio*3.1416
3    perímetro = 2*radio*3.1416
4    Escribir("El área es ", área, " m2",NL)
5    Escribir("El perímetro es ", perímetro, " m")
```

(b) Las instrucciones escritas en **(a)** hacen uso de tres variables: `radio`, `área` y `perímetro`. La primera es una variable de entrada del algoritmo ya que almacena el dato ingresado por el usuario al ejecutarse el paso **1**, mientras que las otras dos son variables de salida, ya que almacenan los resultados del procesamiento de los pasos **2** y **3**, los cuales son mostrados al usuario en los pasos **4** y **5**.

(c) Para simular la ejecución realizamos la "traza" del algoritmo, esto es una tabla de doble entrada con una instrucción por fila y una columna por cada variable que va mostrando la evolución de cada dato a medida que progresa la ejecución del algoritmo. Incluimos también una columna que muestra la evolución de la salida por pantalla.

Paso	`radio`	`área`	`perímetro`	Pantalla
0	?	?	?	`Ingrese el radio en metros:`
1	1.5	?	?	`Ingrese el radio en metros: 1.5`
2	1.5	7.069	?	`Ingrese el radio en metros: 1.5`
3	1.5	7.069	9.425	`Ingrese el radio en metros: 1.5`
4	1.5	7.069	9.425	`Ingrese el radio en metros: 1.5` `El área es 7.069 m2`
5	1.5	7.069	9.425	`Ingrese el radio en metros: 1.5` `El área es 7.069 m2` `El perímetro es 9.425 m`

Representación de Algoritmos

Durante la fase de diseño de los algoritmos, se cuenta con distintas herramientas para poder representarlos, a continuación se abordarán dos de estas: los *diagramas de flujo* y el *pseudocódigo*.

Estas herramientas nos permiten documentar el diseño e intercambiar ideas con colegas. Al mismo tiempo, contar con un algoritmo correctamente representado permite analizar y predecir el comportamiento que tendrá el programa antes de codificarlo y así detectar errores en una etapa temprana del desarrollo.

Pseudocódigo

El pseudocódigo permite una representación de alto nivel de abstracción, donde usualmente se ocultan detalles de implementación irrelevantes para la comprensión del algoritmo, combinando texto común con estructuras y construcciones similares a las de los lenguajes de programación. De una manera distendida, los algoritmos en los ejemplos 2.1 y 2.2 están representados en pseudocódigo, utilizando líneas de texto para representar cada paso o instrucción a ejecutarse en el algoritmo, ordenadas según la secuencia de ejecución.

Si bien no hay una sintaxis estándar para el pseudocódigo, los algoritmos se escriben en forma de listado de instrucciones ordenadas según su secuencia de ejecución y además suelen contar con una sección opcional donde se enumeran los datos (variables y/o constantes) involucrados en la ejecución del algoritmo, así como su tipo. A su vez pueden indicarse los puntos de inicio y de finalización del pseudocódigo, junto con un nombre que lo represente.

Por ejemplo, en el código 2.1 se puede observar el algoritmo del ejemplo 2.2, representado en un pseudocódigo más detallado.

```
INICIO Área_y_Perímetro
    Datos:
       Variables:
          reales: radio, área, perímetro
       Constantes:
          real: PI = 3.141592653
    Algoritmo:
       Escribir(“Ingrese el radio en metros: “)
       Leer(radio)
       área = radio*radio*PI
       perímetro = 2*radio*PI
       Escribir(“El área es “, área, “ m2”,NL)
       Escribir(“El perímetro es “, perímetro, “ m”)
FIN
```

Código 2.1: *Algoritmo del ejemplo 2.2.*

Ejemplo 2.3:

Escriba la representación en pseudocódigo del método numérico iterativo que calcula la raíz cuadrada de un número. El dato de entrada será ingresado por el usuario en la variable `x`. El resultado será mostrado en pantalla.

Método Numérico:

Paso 1- en la variable `r` que se utilizará para devolver el resultado se asigna inicialmente `x`

Paso 2- mientras que `r*r` sea diferente de `x` (aún no se halló la raíz de `x`) se repite el Paso 3

Paso 3- se actualiza el valor de `r` como el promedio entre su valor actual y `x/r`

Paso 4- la raíz cuadrada de `x` quedó almacenada en `r`

Resolución:

Se comienza por los indicadores de inicio y fin, sabiendo que habrá solo dos variables, `x` y `r`, se indican en la sección de datos. Nótese que entre las marcas `/*` y `*/` se escriben comentarios, los cuales tienen aclaraciones que ayudan a la comprensión del algoritmo pero no son instrucciones ejecutables.

```
INICIO Raíz_cuadrada
    Variables:
       reales: x, r
    Algoritmo:
       /*
         pendiente
       */
FIN
```

En segundo lugar, se sabe que `x` será la variable de entrada ingresada por teclado, y `r` la variable de salida a mostrar en pantalla.

```
INICIO Raíz_cuadrada
    Variables:
       reales: x, r
    Algoritmo:
       Leer(x)
       /*
        método numérico r <-raiz(x)
       */
       Escribir(r, “es la raíz cuadrada de ”, x)
FIN
```

Solo resta escribir las instrucciones correspondientes al método numérico como expresiones algorítmicas, teniendo en cuenta que el paso 4 no es necesario implementarlo.

```
INICIO Raíz_cuadrada
    Variables:
       reales: x, r
    Algoritmo:
       Leer(x)
       r = x                        /* Paso 1 */
       Mientras r*r <> x repetir:     /* Paso 2 */
          r = (x/r + r)/2           /* Paso 3 */
       Fin_Mientras       /* esto indica el fin de la repetición*/
       Escribir(r, “es la raíz cuadrada de ”, x)
FIN
```

Nótese que los pasos 2 y 3 se ejecutarán varias veces hasta que `r*r` sea igual a `x`, y que `Fin_Mientras` se usó para delimitar las instrucciones que se repetirán mientras `r*r` sea distinto de `x`. Esta construcción forma parte de las *estructuras de control* propias de la *programación estructurada.*

Diagrama de flujo

El diagrama de flujo (DF) representa gráficamente un algoritmo, mediante símbolos estandarizados por la ANSI y la ISO, facilitando la identificación de los distintos caminos o flujos de ejecución y la toma de decisiones.

Por ejemplo, en la figura 2.4 se representa el DF del ejemplo 2.3, que permite calcular la raíz cuadrada de un número ingresado por el usuario. En dicha figura se explican los símbolos utilizados.

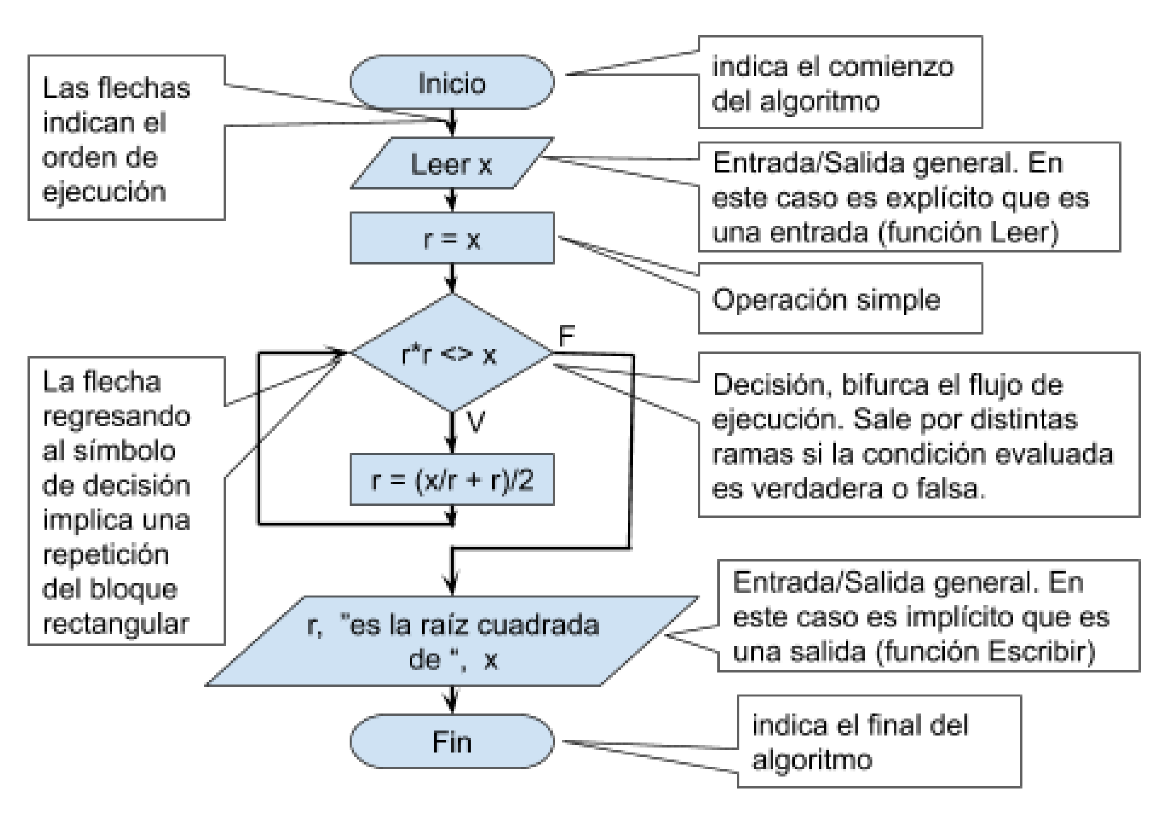


Figura 2.4: *Algoritmo del ejemplo 2.3 representado mediante DF.*

A continuación, la tabla 2.7 describe detalladamente todos los símbolos que se utilizarán para realizar diagramas de flujo.

Tabla 2.7: Algunos de los símbolos estandarizados para diagramas de flujo.

↓ →	**Flechas:** cualquier flecha en un DF indicará el *orden temporal* en que se ejecutarán distintas acciones (*flujo de ejecución*). Suelen graficarse con orientación descendente, con lo cual el DF comienza en su parte superior y finaliza en la inferior.
Inicio	**Inicio del algoritmo:** Aparece una única vez en el DF en su parte superior. No tiene ninguna flecha de ingreso y una única flecha de salida que dirige el flujo de ejecución hacia la primera instrucción del algoritmo. palabra “Inicio” en su interior, el nombre del algoritmo o *función* y parámetros de entrada/salida en el caso de *funciones*.
Fin	**Fin del algoritmo:** Aparece al menos una vez en el DF, en la parte inferior del mismo. Tiene una flecha de ingreso y ninguna flecha de salida. Suele contener la palabra “Fin” en su interior.
	Proceso: Operación *definida* o conjunto de operaciones. contienen *expresiones* simples (compuestas por *operadores* y *operandos*) y *asignaciones* que manipulan la información.
	Proceso definido: Ejecuta una subrutina o *función* definida en algún otro algoritmo. En este curso solemos utilizar el rectángulo normal (**Proceso**) en lugar del símbolo específico para el **proceso definido.**
V expresión F	**Decisión**: Dentro del rombo se inscribe la expresión a evaluar. Se toma un camino diferente según el valor de la expresión. Si la misma es lógica hay un camino para el verdadero y otro para el falso, si es numérica los caminos posibles pueden ser varios.
Leer (x) x Escribir(x) x	**Entrada/Salida:** Se representan con paralelogramos que en su interior indican la función de E/S que se ejecutará. Si no se indica el nombre de la función se sobreentiende que se están utilizando las funciones *Leer()* o *Escribir()*. Si bien no está estandarizado, el uso opcional de una flecha en el vértice superior derecho puede ayudar a distinguir si se trata de una entrada o salida.
Comentario sobre alguna parte del diagrama de flujo	**Comentario:** Brinda más información y ayuda a comprender ciertas partes del algoritmo. No se ejecutan.
	Repetición: Se utiliza para representar algunas estructuras de control repetitivas.

Programación estructurada

El paradigma de la programación estructurada es uno de los dos paradigmas de programación sobre los cuales se basa este curso. Este establece que cualquier programa se puede implementar mediante un conjunto acotado de *estructuras de control de flujo* que pueden agruparse en las siguientes categorías: Estructura secuencial, estructuras selectivas y estructuras iterativas.

Esto tiene impacto positivo en la legibilidad del código, en el tratamiento de fallos y mantenimiento, así como en el tiempo de desarrollo.

Estructura secuencial

Es la estructura que se forma naturalmente al establecer que un paso o instrucción se ejecutará a continuación de otro. En la tabla 2.8 se puede apreciar la estructura secuencial en DF y pseudocódigo con tres acciones sucesivas, las cuales pueden representar una simple instrucción o encapsular un conjunto de instrucciones.

Tabla 2.8: Representación de la estructura selectiva simple

Diagrama de flujo	Pseudocódigo
<acción 1> → *<acción 2>* → *<acción N>*	<acción 1> <acción 2> <acción 3>

Estructuras de control selectivas

Son estructuras donde el flujo de ejecución se bifurca según el valor de una expresión. Se consideran en este curso tres tipos de estructuras selectivas diferentes:

Selectiva simple

Evalúan una expresión lógica o condición y sólo en caso de ser verdadera se ejecuta una o un conjunto de acciones, mientras que si es falsa no se ejecuta nada. A la selectiva simple se la conoce como estructura `SI`. En la Tabla 2.9 puede observarse su implementación en pseudocódigo y en diagrama de flujo.

Tabla 2.9: Representación de la estructura selectiva simple

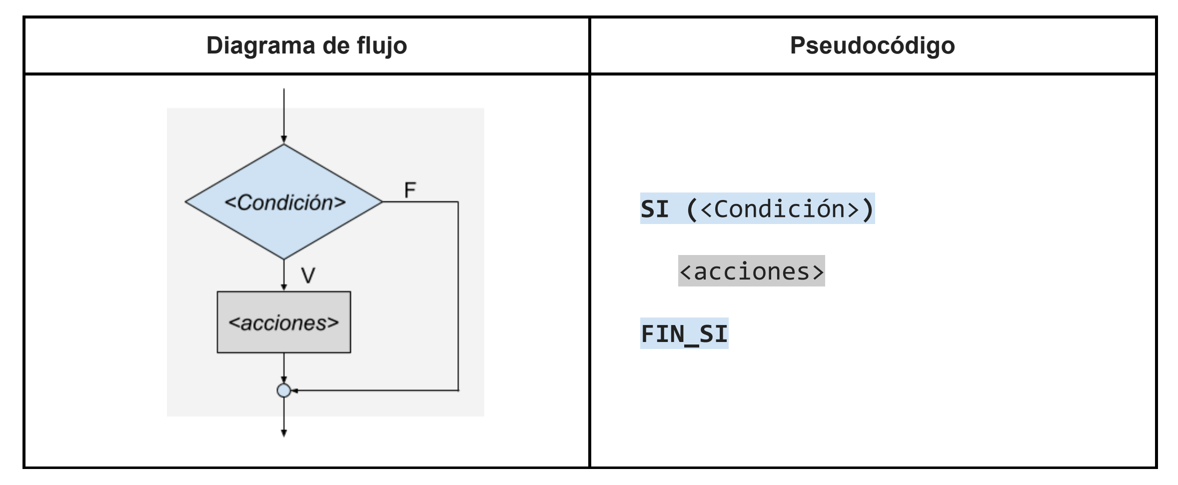


Diagrama de flujo	Pseudocódigo
<Condición> (F / V) → *<acciones>*	`SI (<Condición>)` `    <acciones>` `FIN_SI`

Selectiva doble

Evalúan una expresión lógica o condición y bifurcan el código en dos ramas distintas de ejecución según sea verdadera o falsa la expresión. A la selectiva doble se la conoce como estructura `SI/SINO`. En la Tabla 2.10 puede observarse su implementación en pseudocódigo y en diagrama de flujo.

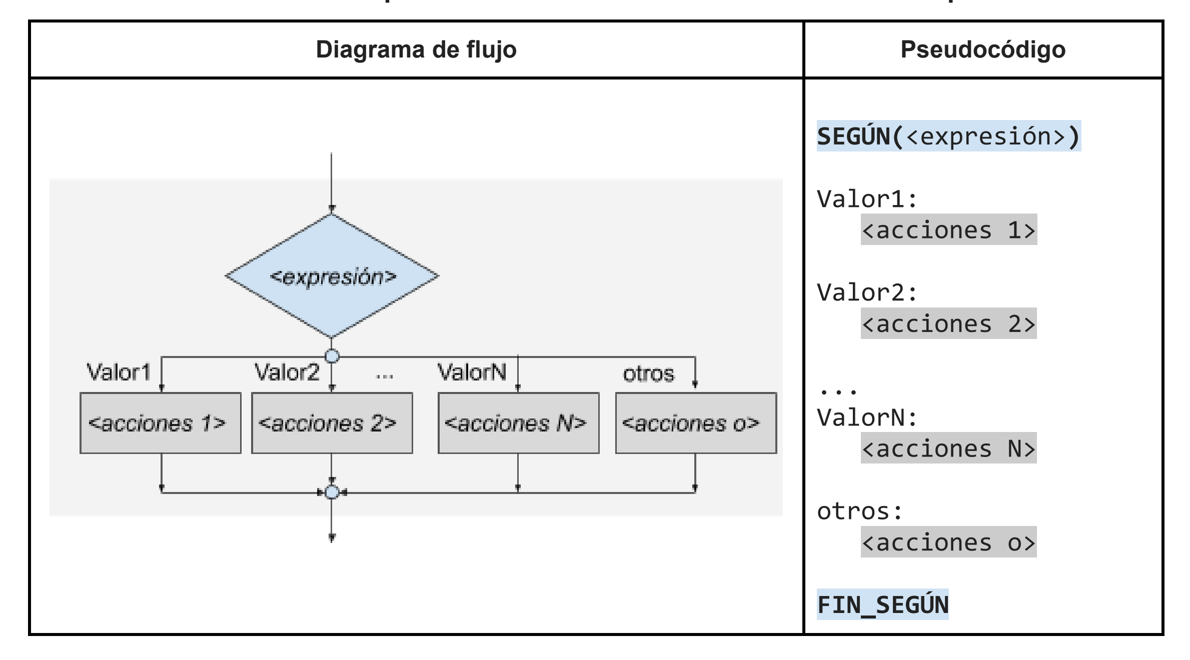
Tabla 2.10: Representación de la estructura selectiva doble

Diagrama de flujo	Pseudocódigo
<Condición> V F <acciones V> <acciones F>	**SI** (<Condición>) <acciones V> **SINO** <acciones F> **FIN_SI**

Selectiva múltiple

Evalúan una expresión entera (o de texto), que puede tomar un conjunto finito de N valores distintos. A la selectiva múltiple se la conoce como estructura `SEGÚN`. En la Tabla 2.11 puede observarse su implementación en pseudocódigo y en diagrama de flujo. La opción `otros` indica el camino de ejecución que se seguirá si el resultado de `<expresión>` no coincide con ninguno de los N valores evaluados.

Tabla 2.11: Representación de la estructura selectiva múltiple

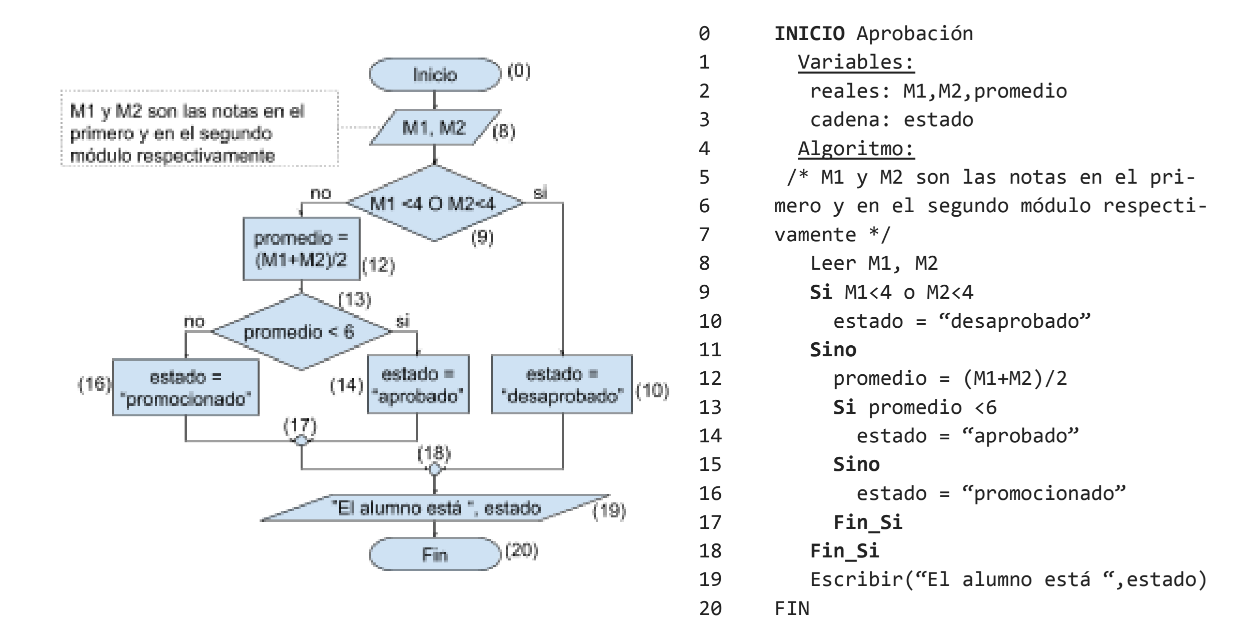
Diagrama de flujo	Pseudocódigo
	**SEGÚN**(<expresión>) Valor1: <acciones 1> Valor2: <acciones 2> ... ValorN: <acciones N> otros: <acciones o> **FIN_SEGÚN**

Ejemplo 2.4:

Representar el algoritmo (en DF y pseudocódigo) de un programa que pide ingresar las notas de un alumno en los dos módulos de una materia. El programa deberá indicar si el alumno está desaprobado (alguna de las notas menor a 4), aprobado (ambas notas mayor o igual a 4 y promedio menor a 6) o promocionado (ambas notas mayor o igual a 4 y promedio mayor o igual a 6).

Resolución:

Se observa a continuación el algoritmo resultante, representado tanto en DF como en pseudocódigo, que consta de dos estructuras selectivas dobles. La primera en evaluarse detecta un caso de desaprobación si alguna de las dos notas es menor a 4. De ser esto verdadero, se le asigna la cadena `“desaprobado”` a la variable de salida `estado`, se sale de la estructura selectiva y se ejecuta la última instrucción del algoritmo que muestra el resultado en pantalla. En caso de ser falsa la condición mencionada, habrá que distinguir entre los estados de aprobación y promoción, y para esto hace falta una segunda estructura selectiva doble, que está *anidada* dentro de la primera. Con fines exclusivamente didácticos se indica con números en el DF la línea equivalente en el pseudocódigo.




```
0    INICIO Aprobación
1      Variables:
2        reales: M1,M2,promedio
3        cadena: estado
4      Algoritmo:
5     /* M1 y M2 son las notas en el pri-
6    mero y en el segundo módulo respecti-
7    vamente */
8        Leer M1, M2
9        Si M1<4 o M2<4
10         estado = “desaprobado”
11       Sino
12         promedio = (M1+M2)/2
13         Si promedio <6
14           estado = “aprobado”
15         Sino
16           estado = “promocionado”
17         Fin_Si
18       Fin_Si
19       Escribir(“El alumno está “,estado)
20   FIN
```

Ejemplo 2.5:

Representar el algoritmo en pseudocódigo de un programa que muestre en pantalla un menú con 4 opciones:

1) sumar 2 números
2) restar 2 números
3) multiplicar 2 números
4) dividir 2 números

El programa deberá pedir al usuario que ingrese la opción deseada y luego los valores con los cuales realizar la operación elegida. Por simplicidad, se asume que el usuario siempre ingresará una opción válida.

Resolución:

El problema propuesto puede resolverse mediante el uso de una selectiva múltiple que en función de la opción elegida realice la operación aritmética correspondiente.

```
INICIO
    Variables:
       reales: a,b,res
       entero: opción
    Algoritmo:
       Escribir “1)sumar 2 números”,NL
       Escribir “2)restar 2 números”,NL
       Escribir “3)multiplicar 2 números”,NL
       Escribir “1)dividir 2 números”,NL
       Escribir “Ingrese la opción deseada: ”
       Leer opción
       Leer a,b

       Según(opción)
          1:
             res = a + b
          2:
             res = a - b
          3:
             res = a * b
          4:
             res = a / b
       Fin_Según
       Escribir(“El resultado es “, res)
FIN
```

Estructuras de control iterativas

Son las estructuras que permiten repetir la ejecución de determinada parte del código. Dentro de las estructuras iterativas trabajaremos con tres variantes distintas.

Mientras

Evalúa una expresión lógica o condición y en caso de ser verdadera se ejecuta una o un conjunto de acciones, luego de dicha ejecución se vuelve a evaluar la condición mencionada, produciendo que mientras dicha expresión lógica sea verdadera, las instrucciones contenidas por la estructura de control se repitan. Cuando el resultado de la expresión lógica es falso se sale de la estructura de control.

En la Tabla 2.12 puede observarse su implementación en pseudocódigo y en diagrama de flujo.

Tabla 2.12: Representación de la estructura iterativa Mientras

Diagrama de flujo	**Pseudocódigo**
<Condición> F V <acciones>	**MIENTRAS (*<Condición>*)** <acciones> **FIN_MIENTRAS**

Hacer-Mientras

Primero ejecuta una o un conjunto de acciones, contenidas en la estructura de control, y luego se evalúa una expresión lógica o condición. En caso de ser verdadera se ejecuta nuevamente el conjunto de acciones, produciendo que mientras dicha expresión lógica sea verdadera, las instrucciones contenidas por la estructura de control se repitan. Cuando el resultado de la expresión lógica es falso se sale de la estructura de control.

En la Tabla 2.13 puede observarse su implementación en pseudocódigo y en diagrama de flujo.

Tabla 2.13: Representación de la estructura iterativa Hacer-Mientras

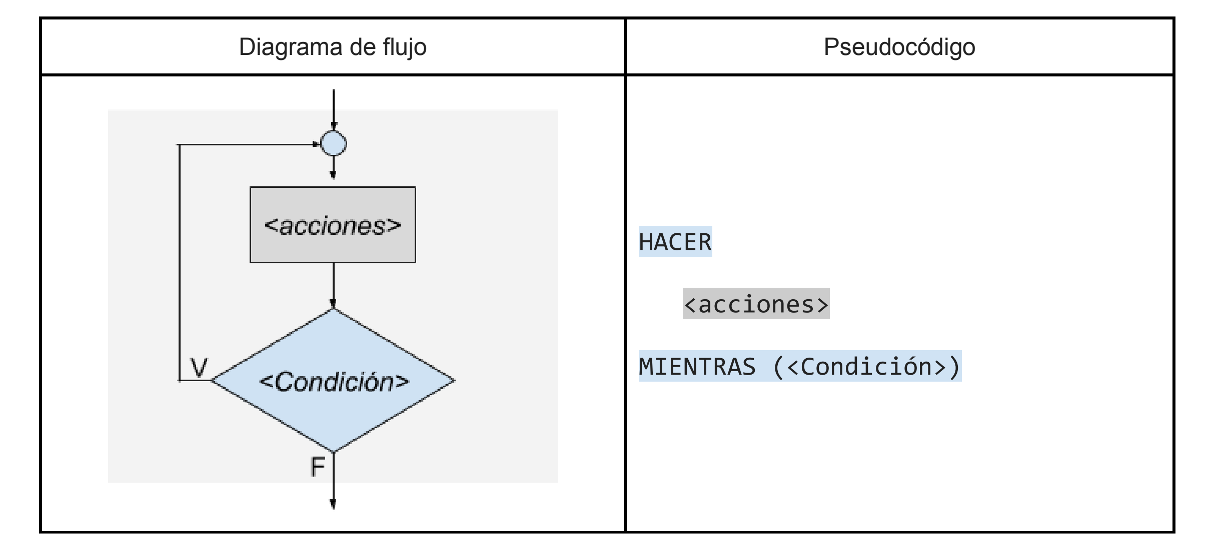


Diagrama de flujo	Pseudocódigo
<acciones> V <Condición> F	HACER <acciones> MIENTRAS (<Condición>)

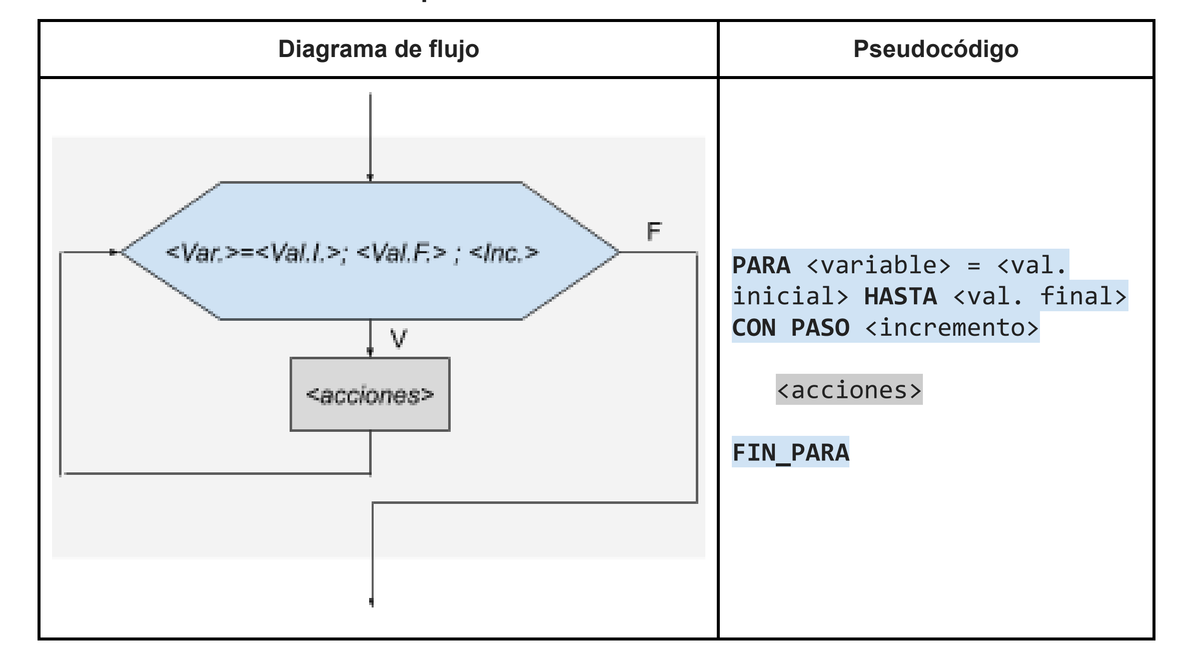
La diferencia entre las estructuras `MIENTRAS` y `HACER-MIENTRAS` está en que en el `MIENTRAS`, primero se verifica la condición y solo si esta es verdadera se ejecutan las acciones contenidas, pero en el `HACER-MIENTRAS` primero se ejecutan las acciones y luego se verifica la condición para repetir las acciones. Por lo tanto, el `HACER-MIENTRAS` asegura que al menos una vez las acciones contenidas serán ejecutadas, a diferencia del `MIENTRAS`.

Para

La estructura `PARA` suele utilizarse para repetir una o un conjunto de acciones una cantidad de veces conocida. Tiene asociada una variable que se utiliza para controlar las iteraciones, para la cual se indica el valor inicial, el valor final y el incremento o decremento que esta sufre entre repeticiones.

En la Tabla 2.14 puede observarse su implementación en pseudocódigo y en diagrama de flujo.

Tabla 2.14: Representación de la estructura iterativa Para

Diagrama de flujo	Pseudocódigo
	**PARA** <variable> = <val. inicial> **HASTA** <val. final> **CON PASO** <incremento> <acciones> **FIN_PARA**

Ejemplo 2.6:

Realice el pseudocódigo y el DF de un programa que escriba en pantalla los números pares entre 2 y 10, utilizando:

a) MIENTRAS
b) HACER-MIENTRAS
c) PARA

Resolución:

a)

```
INICIO
    Variables:
       entero: i
    Algoritmo:
       i=2
       Mientras i <= 10
          Escribir(i,NL)
          i=i+2
       Fin_Mientras
FIN
```

b)

```
INICIO
    Variables:
       entero: i
    Algoritmo:
       i=2
       Hacer
          Escribir(i,NL)
          i=i+2
       Mientras i <= 10
FIN
```

c)

```
INICIO
    Variables:
       entero: i
    Algoritmo:
       Para i=2 hasta 10 con paso 2
          Escribir(i,NL)
       Fin_Para
FIN
```

Ejemplo 2.7:

Repita los pseudocódigos del ejemplo 2.6 pero ahora imprima los números pares entre 2 y `N`, siendo `N` un valor ingresado por el usuario.

Resolución:

Se muestran en rojo las modificaciones realizadas a la resolución del ejemplo 2.6

a)

```
INICIO
    Variables:
       entero: i,N
    Algoritmo:
       Leer(N)
       i=2
       Mientras i <= N
          Escribir(i,NL)
          i=i+2
       Fin_Mientras
FIN
```

b)

```
INICIO
    Variables:
       entero: i,N
    Algoritmo:
       Leer(N)
       i=2
       Hacer
          Escribir(i,NL)
          i=i+2
       Mientras i <= N
FIN
```

c)

```
INICIO
    Variables:
       entero: i,N
    Algoritmo:
       Leer(N)
       Para i=2 hasta N con paso 2
          Escribir(i,NL)
       Fin_Para
FIN
```

Programación modular

La programación modular es un paradigma de programación que también aplicaremos durante este curso, además de la ya mencionada programación estructurada.

Este paradigma se basa en la aplicación del principio “divide y reinarás” a problemas complejos, los cuales son divididos en un conjunto de problemas más sencillos. Cada uno de dichos problemas sencillos es considerado un “módulo”, el cual es resuelto individual-

mente mediante un algoritmo particular. La solución al problema complejo más general resulta entonces de juntar todos los "módulos" y ejecutar dichos algoritmos coordinadamente desde un *programa principal.*

Son varias las ventajas de aplicar el paradigma de la programación modular, entre ellas:

- Promueve el diseño *top-down*, permitiendo el diseño de soluciones generales que hacen uso de bloques abstractos, que son implementados posteriormente.
- Facilita el trabajo en equipo, haciendo que cada programador se centre en la resolución de un problema particular.
- Permite la reutilización de código.
- Facilita el análisis del código, su mantenimiento y depuración de errores.

Concepto de función

La herramienta que brindan los distintos lenguajes de programación para implementar los mencionados "módulos" se denomina *función*, aunque también suelen usarse los nombres *procedimiento, subrutina* o *subprograma*.

Como se muestra en la figura 2.5, cada función será un algoritmo, según lo hemos definido anteriormente, que puede recibir datos de entrada (parámetros o argumentos) y entregar datos de salida como resultado del procesamiento (valor de retorno).

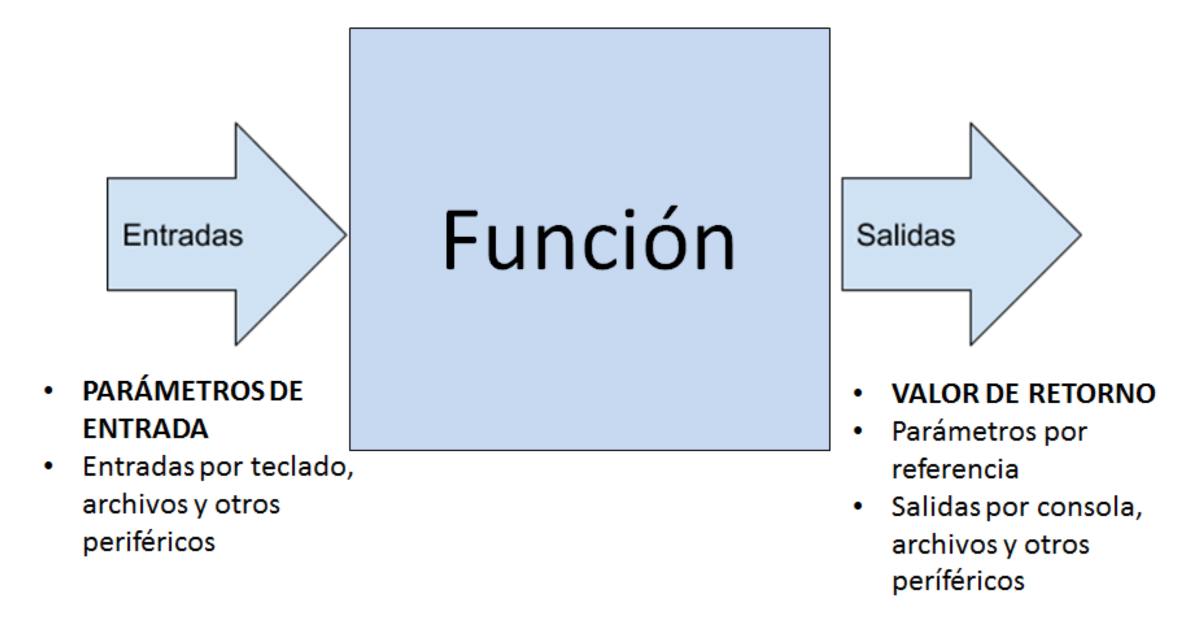


Figura 2.5: *Las funciones implementan algoritmos que, además de las entradas y salidas típicas, cuentan con entradas (parámetros) y salidas (parámetros/valor de retorno) especiales.*

Una vez definido el algoritmo de una función, esta puede ser *utilizada* (*ejecutada, invocada o llamada*) desde otras partes del programa, cuantas veces se lo desee.

Definición de una función

Simplemente para distinguir el algoritmo de un programa principal de los demás sub-algoritmos o funciones, usaremos las palabras clave `FUNCIÓN` y `FIN_FUNCIÓN` para definir a estas últimas. Así mismo las funciones deben tener un *nombre* que las identifique y una lista de los parámetros o variables de entrada, si los hubiese.

```
FUNCIÓN <nombreFunción> (lista de parámetros formales)
   Variables:
      <variables locales>
   Algoritmo:
      <acciones>
      RETORNAR <valor de retorno opcional>
FIN_FUNCIÓN
```

Funciones sin valor de retorno

Las funciones que no retornan un valor no incluyen la instrucción con la palabra clave `RETORNAR`, esto significa que cuando sean utilizadas no podrán formar parte de una expresión más compleja, sino que realizarán un procedimiento simple. En el código 2.2 puede verse la definición de dos funciones `Saludar` y `Saludar_Mucho`, que no retornan ningún valor. La primera no recibe ningún parámetro de entrada y la segunda un valor entero que modifica el comportamiento de la función.

```
FUNCION Saludar()
  Algoritmo:
    Escribir("Hola")
FUNCION

FUNCION Saludar_Mucho(entro: veces)
  Variables:
    entero:i
  Algoritmo:
    Para i=1 Hasta veces
       Escribir("Hola",NL)
    Fin_Para
FIN_FUNCION
```

Código 2.2: *Definición de dos funciones sin valor de retorno*

Funciones con valor de retorno

Si la función devolviese algún valor como resultado, se utiliza la palabra clave `RETORNAR` acompañada del valor de salida correspondiente, la cual hace que finalice el algoritmo de la función.

En el código 2.3 se muestra el algoritmo que calcula la raíz cuadrada de un número (ejemplo 2.3) convertido en una función. Pero, a diferencia del programa original, la función no tiene interacción con el usuario, sino que recibe la variable `x` como un parámetro de entrada, mientras que la raíz calculada en la variable `r` se retorna como resultado de la función.

```
FUNCION Raíz_cuadrada(real: x)
    Variables:
       real: r
    Algoritmo:
       r = x
       Mientras r*r <> x
          r = (x/r + r)/2
       Fin_Mientras

       RETORNAR r
FIN_FUNCION
```

Código 2.3: *Definición de la función Raíz_cuadrada()*

Uso de una función (llamada)

Para usar una función desde otro algoritmo solo se debe escribir su nombre acompañado por un par de paréntesis con los valores pasados como parámetros, a esto se lo conoce como la *invocación* o *llamada* a la función. Con la llamada a la función comienza a ejecutarse su algoritmo y una vez finalizado el mismo el flujo de ejecución retorna al algoritmo desde el cual se hizo la llamada.

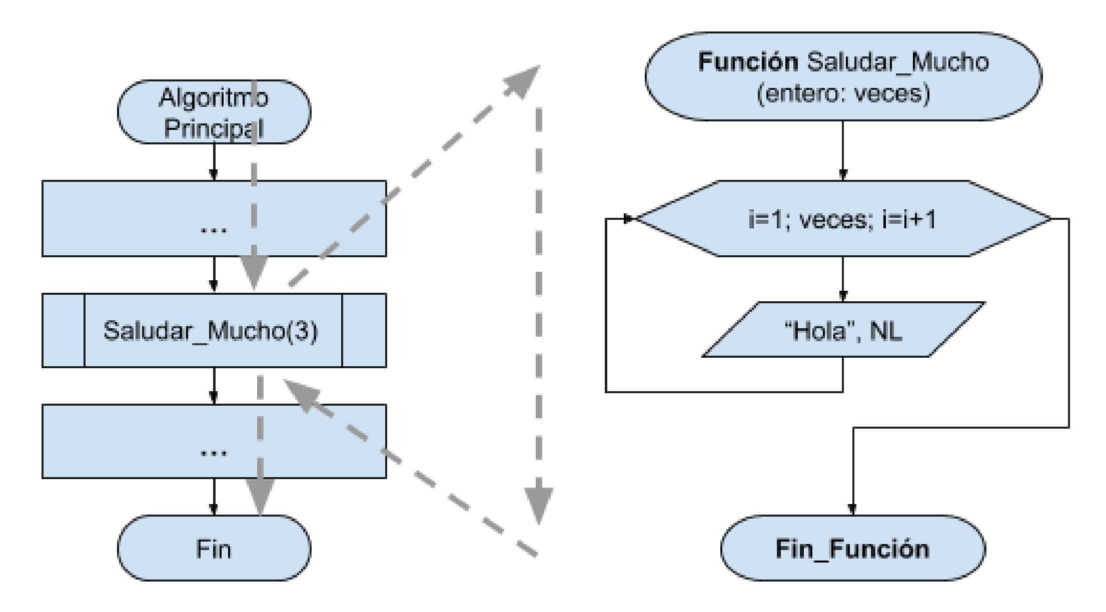


Figura 2.6: *Ejemplo de llamada a una función y saltos en el flujo de ejecución, para la función definida en el pseudocódigo del código 2.2.*

En la figura 2.6 puede verse un ejemplo del llamado a la función `Saludar_Mucho`, definida en el código 2.2, desde el algoritmo principal. Esto hará que el flujo de ejecución “salte” del algoritmo principal al algoritmo de la función `Saludar_Mucho`. Dicho algoritmo se ejecutará con su parámetro de entrada `veces` tomando el valor 3 y al finalizar el mismo se continuará ejecutando el algoritmo principal.

Las funciones que no reciben parámetros de entrada se invocan dejando los paréntesis vacíos, como en el caso de la función `Saludar` del código 2.2, cuya invocación sería:

```
Saludar()
```

Uso del valor de retorno de una función

Cuando se invoca una función que retorna un valor mediante la palabra clave `RETORNAR` se puede hacer uso del valor por la función, si la misma es invocada dentro de una expresión más compleja que hace uso de dicho valor.

Por ejemplo, para calcular la raíz cuadrada de 121 en cualquier parte de un programa, usando la función definida en el código 2.3, será necesaria la siguiente línea de código:

```
Raíz_cuadrada(121)
```

Esto hará que se invoque la ejecución del algoritmo definido en el código 2.3 el cual se ejecutará de inicio a fin, asignando a la variable de entrada `x` el valor 121 pasado como parámetro. Una vez finalizada la ejecución de la función, la llamada es reemplazada por el valor retornado (11), por lo que la instrucción anterior por si sola no tiene ninguna utilidad, pero podría utilizarse como parte de otras expresiones, por ejemplo:

```
y = Raíz_cuadrada(121)           /* asigna 11 a una variable y */
Escribir(Raíz_cuadrada(121))     /* muestra el 11 en pantalla */
```

Funciones con más de un parámetro

Las funciones pueden tener más de un parámetro de entrada. Esto se indica por un lado en el momento de la definición, separando por coma cada parámetro formal dentro del paréntesis, así como en la invocación de la función a la cual deben pasársele todos los entre paréntesis tantos valores separados por coma como parámetros se hayan indicado.

En el ejemplo del código 2.4 puede observarse la función `Hipotenusa` que recibe dos parámetros, la `base` y la `altura` de un triángulo rectángulo, y devuelve la hipotenusa del mismo.

```
INICIO
    Variables:
       reales: b,a
    Algoritmo:
       Escribir("Ingrese base y altura:")
       Leer(b)
       Leer(a)
       Escribir("La hipotenusa es ",Hipotenusa(b,a))
FIN

FUNCION Hipotenusa(real: base, real: altura)
    RETORNAR Raíz_cuadrada(base^2 + altura^2)
FIN_FUNCION
```

Código 2.4: Ejemplo de función con más de un parámetro

Ejercicios

Expresiones Aritmético-Lógicas

1) Escriba las tablas de verdad de las operaciones lógicas Y (AND), O INCLUSIVO (OR) y O EXCLUSIVO (XOR).

2) Utilizando los operadores relacionales y lógicos escriba expresiones que resulten verdaderas o falsas para las siguientes propuestas:
   a) Verificar si una variable numérica x tiene un valor entre 17 y 135
   b) Verificar si una variable numérica x NO tiene un valor entre 17 y 135 (hacerlo de dos maneras distintas)
   c) Verificar si una temperatura t está por debajo del punto de congelamiento o por encima del de ebullición del agua.

3) Obtener el resultado (verdadero o falso) de las siguientes expresiones lógicas
   a) x >= -5 y x < 14   (considere que x vale -3)
   b) 25>=7 y no (7<=2)
   c) (10>=5 o 23=13) y no (8=8)
   d) (no(6/3>3) o 7>7) y (3 >=9/2 o 2+3>=7/2)

4) Obtener el resultado numérico de las siguientes expresiones teniendo en cuenta el orden de precedencia de los operadores.
   a) 7*10-5mod3*4+9
   b) 5*(5+(6-2)+1)
   c) 7-6/3+2*3/2-4/2

   d) (7*3-4*4)^2/4*2
   e) 7^2*((10-5)mod3)+4*9

5) Convertir las siguientes expresiones algebraicas en algorítmicas usando el menor número de paréntesis.

a) $$\frac{3a+b}{c-\dfrac{d+5e}{f+\dfrac{g}{2h}}}$$

b) $$(a+b)^2-\frac{3t}{h+j}-7k$$

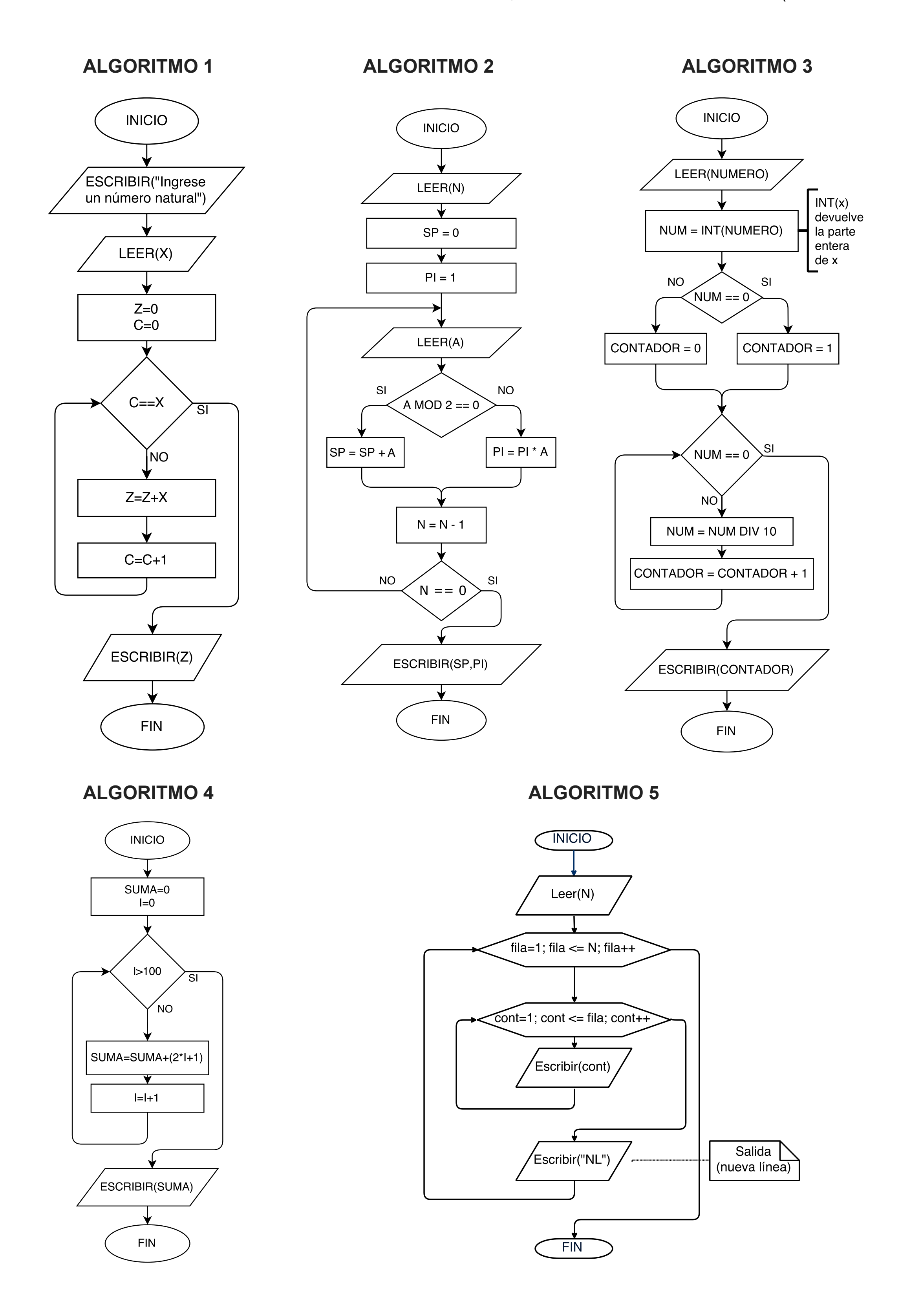
6) Escribir las expresiones para:
   a) Comprobar si una variable x está comprendida en el intervalo [-3,1).
   b) Comprobar su una variable z es positiva y par
   c) Comprobar si una variable j es divisible por 3 y por 4 a la vez

Tipos de datos

7) En los algoritmos la información se almacena en variables. Las variables pueden ser de tipo ENTERO, REAL, CARACTER, CADENA DE CARACTERES, entre otros. ¿Con qué tipo de variable modelaría los siguientes datos y por qué?
   a) Una temperatura
   b) La cantidad de autos que pasan por un detector
   c) El nombre y contraseña de un usuario
   d) Un monto de dinero
   e) El sexo de un usuario

Diagramas de flujo y Pseudocódigo

8) Para cada uno de los siguientes diagramas de flujo:
   a) Enumere los datos de entrada y de salida
   b) Analice y explique la finalidad del algoritmo. Note que en algunos casos necesitará realizar la traza del algoritmo, mientras que para otros esto no resultaría práctico dada la cantidad de iteraciones. Finalmente, en algunos puede ser conveniente reproducir en la hoja la salida que se obtendría en pantalla.



9) Escriba el pseudocódigo correspondiente para los algoritmos 1, 2 y 5.

10) Dados los siguientes programas en pseudocódigo

   a) Enumere los datos de entrada, de salida y auxiliares

   b) Analice y explique la finalidad del algoritmo.

ALGORITMO 1

```
INICIO
    DATOS:
  Variables:
         x, y: real
         n, i: entero
ALGORITMO:
  Leer x, n
  y=1
  Para i=1 hasta abs(n), Inc 1
     y = y * x
  Fin_Para
  Si n<0
                y = 1/y
  Fin_Si
  Escribir y
FIN
```

ALGORITMO 2

```
INICIO
 DATOS:
  Variables:
  borde, figura, opcion: caracter
  f, c, ancho: entero
 ALGORITMO:
  Hacer
   Leer borde, figura, ancho
   Si(ancho mod 2 == 0)
    ancho = ancho-1
   Fin_Si

   Escribir "NL"
   Para c=1, hasta ancho, Inc 1
    Escribir borde
   Fin_Para

   Para f=1, hasta ancho/2, Inc 1
    Escribir "NL"
    Escribir borde
    Para c=1, hasta ancho/2-f, Inc 1
     Escribir " "
    Fin_Para
    Para c=1, hasta 2*f-1, Inc 1
     Escribir figura
    Fin_Para

    Para c=1, hasta ancho/2-f, Inc 1
     Escribir " "
    Fin_Para
    Escribir borde
   Fin_Para

   Escribir "NL"
   Para c=1, hasta ancho, Inc 1
    Escribir borde
   Fin_Para

   Escribir "Desea continuar (S/N)?"
   Leer opcion

  Mientras(opcion != 'N')

FIN
```

11) Dibuje el diagrama de flujo correspondiente para cada uno de los pseudocódigos anteriores.

12) Dado un sistema de ecuaciones lineales

$ax+by = c$

$dx+ey =f$

Realice el diagrama de flujo y el pseudocódigo de un algoritmo que:

- Lea los coeficientes a, b, c, d, e, f
- Verifique que el sistema tenga solución
- Resuelva y visualice los valores x e y.

13) El mayor número

a) Diseñar el algoritmo (ordinograma y pseudocódigo) que muestre el mayor de tres números enteros entrados por teclado.
b) Modifique el programa para encontrar el mayor entre 10 números ingresados por teclado.
c) Encuentre ahora el mayor entre N números ingresados por teclado, donde N también será ingresado por el usuario.

14) Diseñar el algoritmo (ordinograma y pseudocódigo) que calcule la media de una serie de números positivos entrados por teclado. El ingreso de un valor igual a cero indicará el final del ingreso de datos.

15) Realizar el diagrama de flujo y pseudocódigo de un algoritmo que visualice el factorial de un número comprendido entre 2 y 20 ingresado por teclado. Verifique que el valor ingresado esté en el rango correcto y, si no lo está, pida es valor de nuevo al usuario hasta que se cumpla la condición.

16) Diseñar el algoritmo que permita, dados tres números, determinar si la suma de cualquier pareja de ellos es igual al tercer número. Si se cumple esta condición deberá imprimir la palabra “iguales” sino “distintos”.

17) Realice un diagrama de flujo para imprimir una figura de N en pantalla del alto **a** indicado por el usuario por teclado. Por ejemplo:

```
xx    x          xx     x
x  x  x          x  x   x
x    xx          x   x  x
x     x          x     xx
                 x      x
```

Para **a** = 4   Para **a**=5

18) Realizar el Diagrama de flujo y escribir el Pseudocódigo del algoritmo que pida al usuario el ingreso de la hora expresada en horas, minutos y segundos e implemente un reloj que indique cada segundo en pantalla la hora con el formato H:M:S. Utilice la instrucción “Esperar 1 segundo” para detener el flujo de ejecución durante 1 segundo.

Algoritmos modulares

19) Módulo de validación de entrada

a) Cree un módulo o función para pedir y validar una entrada numérica de un usuario, y repetir el pedido hasta que el valor sea correcto. El nombre del módulo debe ser “entrada” y debe tener dos valores enteros como argumentos, que representarán el valor mínimo y máximo del rango a validar. Finalmente, el módulo debe retornar el valor ingresado por el usuario, cuando lo haga correctamente.
b) Vuelva a implementar el algoritmo del ejercicio 15 pero ahora utilizando este módulo.

CAPÍTULO 3
Elementos básicos del lenguaje de programación "C"

Pablo A. García

En este capítulo se introduce el concepto de lenguaje de programación, presentando los distintos tipos y niveles de lenguajes, haciendo un relevamiento de las variantes actuales en cuanto a lenguajes disponibles. Se introduce el lenguaje "C", utilizado en el presente curso.

Se presentan los lenguajes compilados e interpretados, describiendo con mayor detenimiento el proceso de compilación. Se presenta el ambiente de desarrollo integrado (por sus siglas en inglés de IDE: integrated development environment).

Usando el clásico ejemplo del "Hola mundo" se describen las partes de un programa en C.

Se presenta una primera aproximación al lenguaje C, identificando variables y tipos de datos básicos, asignaciones, operadores de todo tipo (con su precedencia), cerrando el capítulo con las estructuras de control.

Lenguajes de programación

Definición

Un lenguaje de programación es un conjunto de símbolos (alfabeto) junto a un conjunto de reglas para combinar dichos símbolos que se usan para expresar programas.

Constan de un léxico, una sintaxis y una semántica.

- Léxico: Conjunto de símbolos permitidos o vocabulario.
- Sintaxis: Reglas que indican cómo realizar las construcciones del lenguaje.
- Semántica: Reglas que permiten determinar el significado de cualquier construcción del lenguaje.

En la actualidad existen cientos de lenguajes de programación que permiten escribir distintos programas en distintos ámbitos. Algunos de estos programas pueden ser ejecutados de manera directa por la computadora, mientras que otros requieren de pasos intermedios de traducción antes de poder ser ejecutados. Es así, que se puede hacer una primera clasificación de los lenguajes de programación por nivel.

Clasificación de lenguajes por nivel

En esta clasificación se utilizan tres niveles:

- Lenguajes de bajo nivel: código máquina.
- Lenguajes de nivel medio: ensamblador.
- Lenguajes de alto nivel: C, Pascal, etc.

El lenguaje de bajo nivel, es directamente el lenguaje propio de la máquina. El mismo está íntimamente relacionado con el hardware propio de la computadora, y normalmente está formado por largas secuencias de ceros y unos que indican a la computadora las instrucciones básicas a ejecutar que conforman un programa. El código máquina resulta difícil de manejar para los seres humanos. Por este motivo, con el avance y difusión de las computadoras, se empezaron a utilizar abreviaturas para representar las operaciones básicas de las computadoras, que es lo que se conoce como lenguaje ensamblador. Para traducir las instrucciones escritas en este lenguaje a código máquina surgieron los ensambladores.

Características del Lenguaje máquina y ensamblador:

- Dependientes del procesador.
- Requiere el conocimiento del hardware del procesador.
- Utiliza instrucciones muy elementales, que realizan operaciones básicas.
- Los programas ocupan poca memoria logrando alta velocidad de ejecución.

Es de destacar que los lenguajes de bajo y medio nivel requieren un amplio conocimiento del hardware y además se necesitan muchas instrucciones para llevar a cabo tareas sencillas. Con el objeto de acelerar el proceso de desarrollo de programas, surgieron los lenguajes de alto nivel, que permiten realizar tareas más complejas con instrucciones simples. Es así, que para traducir estos programas de alto nivel a código máquina surgieron los compiladores y los intérpretes.

Desde sus orígenes, las computadoras manejan lógica binaria, dado que resulta más sencillo discernir entre dos estados lógicos posibles que entre más. Por ejemplo, entre los 10 posibles estados que serían necesarios para representar la base completa de nuestro sistema de numeración decimal. En la Figura 3.1 se puede observar gráficamente, cómo los distintos niveles de los lenguajes de programación resultan más inteligibles por las personas y las computadoras.

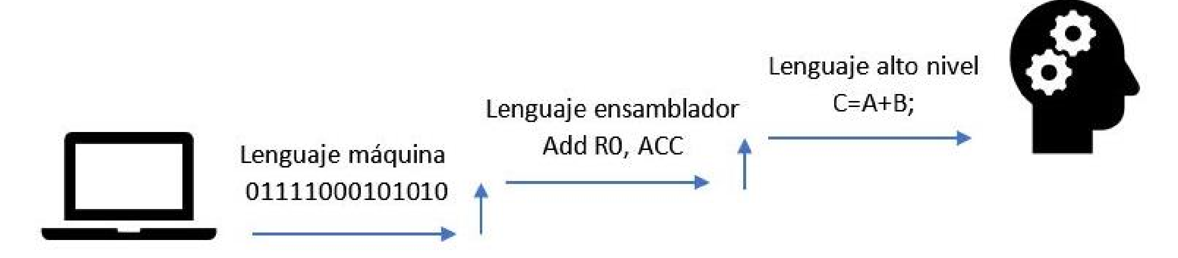


Figura 3.1. *Niveles de lenguajes de programación*

Características de los Lenguajes de alto nivel:

- Independientes del procesador.
- Permiten desconocer detalles del hardware.
- Escritura cercana al lenguaje natural.
- Poseen librerías con funciones de entrada/salida, matemáticas, etc.

Lenguajes interpretados y compilados

En los lenguajes de alto nivel, existe también una clasificación dependiente del tipo de programas que utilizan para traducir el código desde alto nivel a bajo nivel. Cuando este traductor es un intérprete se denominan lenguajes interpretados, mientras que cuando el traductor es un compilador, se denominan lenguajes compilados.

Cuando se usa un lenguaje compilado, la programación se realiza de la siguiente manera:

- Con un editor de texto se escribe el programa fuente.
- Con el compilador se traduce el programa fuente a programa objeto.
- Con el editor de texto se corrigen los errores que devuelva el compilador y se vuelve a compilar, hasta que no haya errores.
- Un enlazador (linker), se aplica al programa objeto y a las funciones de librería utilizadas, para obtener el ejecutable.
- La traducción y ejecución son independientes.

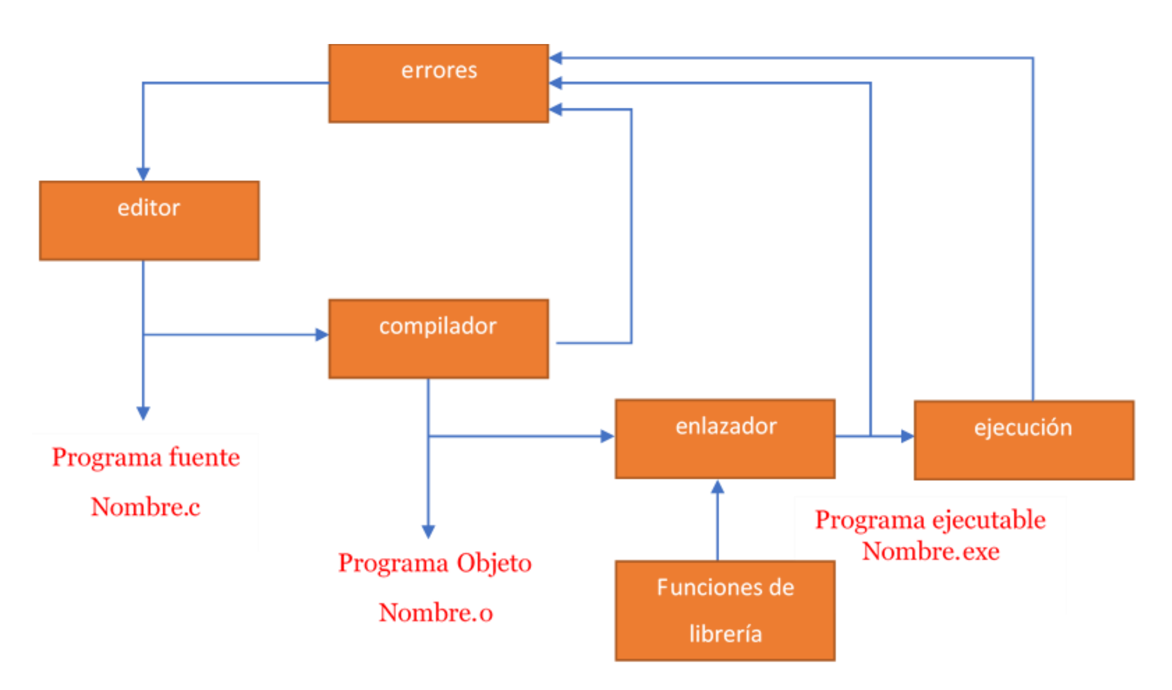


Figura 3.2. *Etapas del diseño con lenguaje compilado "C"*

Cuando se usa un lenguaje interpretado, la programación se realiza de la siguiente manera:

- Con un editor se escribe una instrucción del programa.
- El intérprete inmediatamente traduce la instrucción y la ejecuta, informando de los errores, para que sean corregidos antes de escribir la siguiente instrucción.
- Una vez que el programa ha sido escrito completo sin errores, se almacena, teniendo el código fuente.
- Cada vez que se desee ejecutar el programa, el intérprete irá traduciendo y ejecutando cada instrucción.
- La traducción y ejecución no son independientes.

Ejemplos de lenguajes interpretados: BASIC, JAVA, etc.

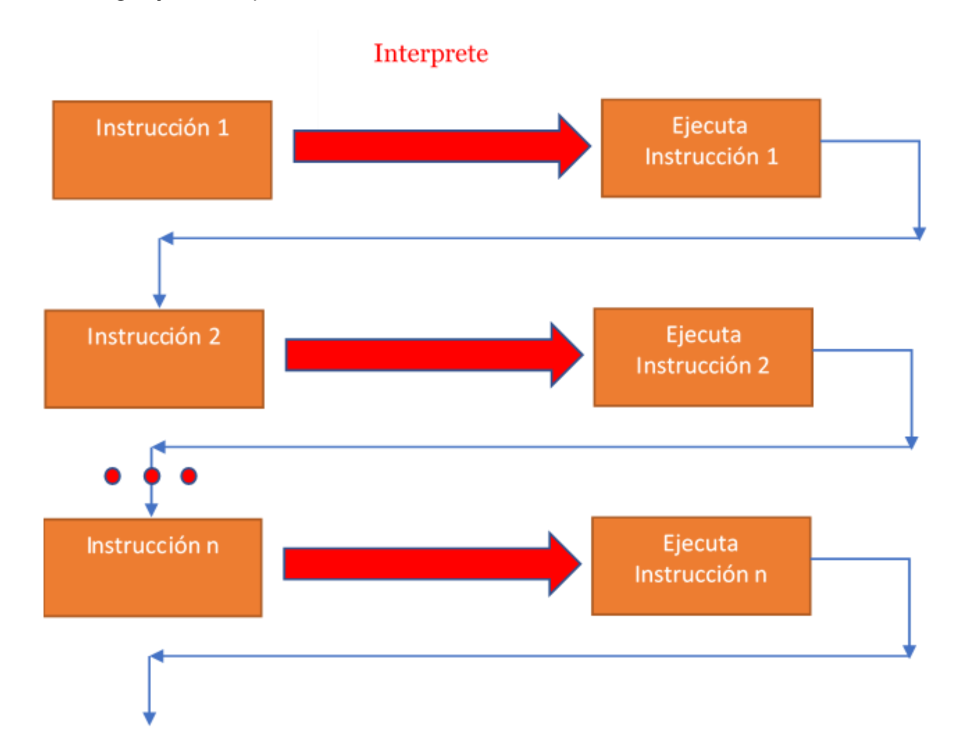


Figura 3.3. *Etapas del diseño con lenguaje interpretado*

Las ventajas de un lenguaje compilado son:

- La ejecución del programa es más rápida, ya que no hay que traducir las instrucciones.
- El programa objeto contiene todas las instrucciones traducidas.
- Durante la ejecución no es necesario tener el compilador en memoria funcionando.

Las ventajas de un lenguaje interpretado son:

- El desarrollo y puesta a punto de un programa es más fácil y rápido, ya que se puede ir corrigiendo los errores a medida que se escriben instrucciones.
- La modificación de programas es más sencilla. No existe un programa objeto almacenado, sólo el fuente.

- Un intérprete ocupa menos memoria que un compilador. Pero debe coexistir con el programa fuente en RAM para su ejecución.

Lenguaje C

C es un lenguaje de programación de propósito general desarrollado como una evolución de los lenguajes B y BCPL. Es un lenguaje procedural, orientado a la implementación de sistemas operativos (originalmente fue UNIX). Este lenguaje se destaca por la eficiencia del código que produce, y es muy popular por permitir crear software de sistema, aunque también se utiliza para crear aplicaciones y software para sistemas embebidos.

En la Figura 3.4 se presenta un gráfico extraído de la web de la comunidad de indexadores TIOBE actualizado a septiembre de 2020, donde puede observarse la permanencia y vigencia del lenguaje C.

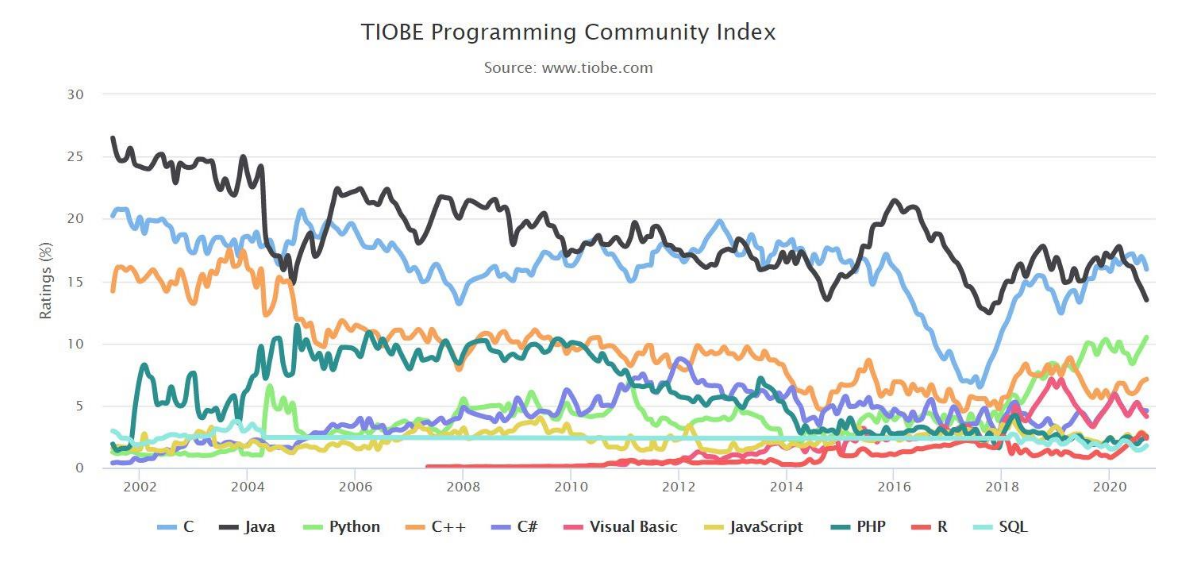


Figura 3.4. *Porcentaje de uso de lenguajes de programación según TIOBE actualizado a septiembre de 2020*

Se trata de un lenguaje de tipos de datos estáticos que dispone de las estructuras típicas de los lenguajes de alto nivel pero, a su vez, dispone de construcciones del lenguaje que permiten un control a muy bajo nivel. Los compiladores suelen ofrecer extensiones al lenguaje que posibilitan mezclar código en ensamblador con código C o acceder directamente a memoria o dispositivos periféricos.

Uno de los objetivos de diseño del lenguaje C es que solo sean necesarias unas pocas instrucciones en lenguaje máquina para traducir cada elemento del lenguaje, sin que haga falta un soporte intenso en tiempo de ejecución.

A pesar de su naturaleza de bajo nivel, el lenguaje se desarrolló para incentivar la programación independiente de la máquina. Un programa escrito cumpliendo los estándares e intentando que sea portátil puede compilarse en muchos computadores [Wikipedia: https://es.wikipedia.org/wiki/C_(lenguaje_de_programaci%C3%B3n)].

Datos de interés:

- Desarrollado entre los años 1969 y 1973 por Dennis Ritchie en los Laboratorios Bell.
- Utilizado para implementar el sistema operativo Unix en la computadora PDP-11.
- En 1978, Brian Kernighan y Dennis Ritchie publican la primera edición de: "The C Programming Language", conocido como "la biblia del C".
- Entre 1983 y 1989 se define el estándar ANSI C.

En 1979, Bjarne Stroustrup agrega a C las clases (C con clases). En 1983 se agregan elementos de la programación orientada a objetos, así nace C++, y en 1985 se publica la primera edición de: "The C++ Programming Language". El lenguaje C++ es muy utilizado en la actualidad, permitiendo tanto el paradigma de programación estructurado y modular (al igual que C), como el paradigma de programación orientado a objetos.

En el año 2000 surge la última implementación de los lenguajes de la familia de C, el C#, que es un lenguaje de programación 100% orientado a objetos desarrollado y estandarizado por Microsoft como parte de su plataforma .NET.

Ensamblador, compilador y enlazador

Normalmente, la compilación de un programa C se realiza en varias etapas que son automatizadas y ocultadas por los entornos de desarrollo; a los cuales conocemos como IDE, por sus siglas en inglés de Integrated Development Environment (Figura 3.5).



Figura 3.5. *Etapas del proceso de compilación en C*

La primer etapa, previa a la compilación, es el Preprocesado en el cual se ejecutan un conjunto de directivas de preprocesado que modifican el código fuente, simplificando de esta forma el trabajo del compilador. Por ejemplo, una de las acciones más importantes es la modificación de las inclusiones (#include) por las declaraciones reales existentes en el archivo indicado y reemplazar las definiciones (#define) por el valor correspondiente.

La siguiente es la etapa de Compilación, en la cual se genera el código objeto a partir del código ya preprocesado. Por último, el Enlazado que une el código objeto de los distintos módulos y bibliotecas externas para generar el programa ejecutable final.

IDE

Un IDE es una aplicación informática que integra varios servicios con el fin de simplificar el trabajo de los programadores. Normalmente, un IDE consiste de un editor de código fuente, herramientas de construcción automáticas (preprocesadores, compiladores y enlazadores) y un depurador o debugger. La mayoría de los IDE tienen autocompletado inteligente de código que resultan muy útiles en tiempo de programación.

En el ámbito de esta cátedra se utiliza el IDE: Code::Blocks. Es un editor para el desarrollo de programas en lenguaje C y C++, distribuido bajo licencia GPL (General Public License). Incluye el compilador gcc.(Figura 3.6).

En el ambiente de trabajo se presentan cuatro áreas bien diferenciadas:

1. Área de trabajo y edición
2. Menú y barras de herramientas.
3. Explorador de proyectos.
4. Área de resultados.

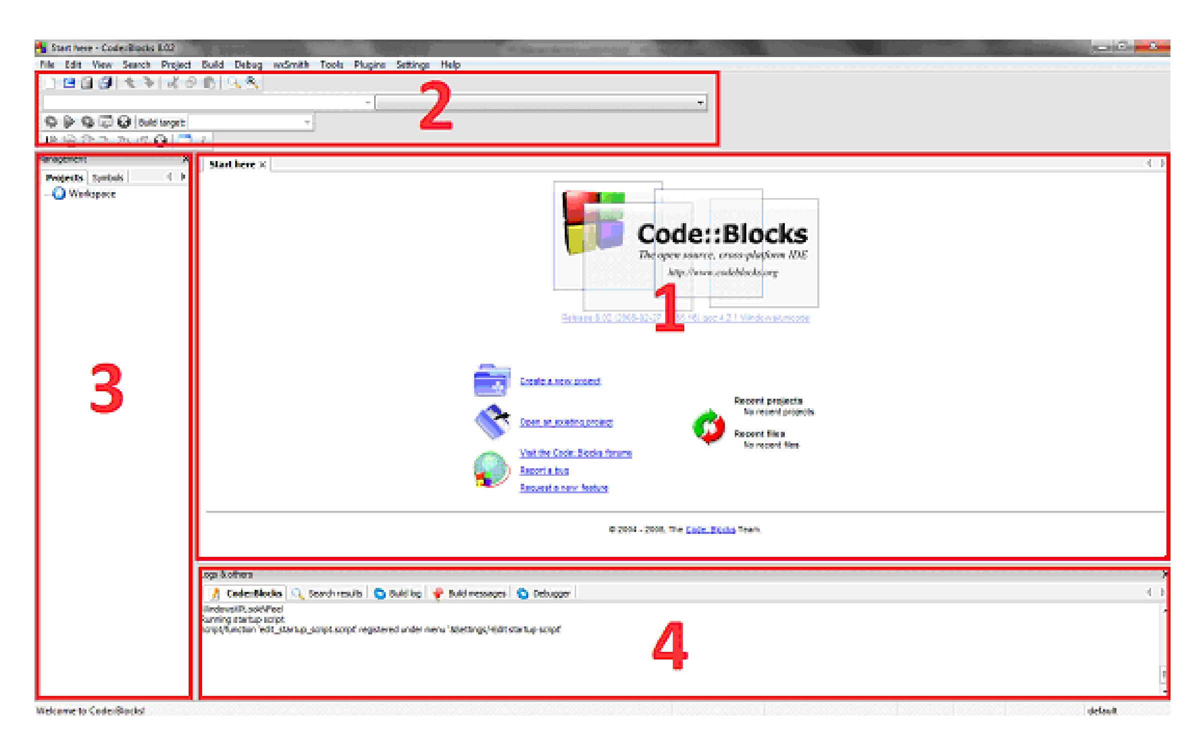


Figura 3.6. *IDE Code::Blocks*

Partes de un programa en C

A continuación, como primer paso en la implementación de programas en lenguaje C, se implementa y describen las partes del clásico primer programa comúnmente denominado “Hola Mundo”, pero con una versión modificada para la cátedra que imprimirá el cartel: “Hola Programación, Algoritmos y Estructuras de Datos”. A continuación se incluye el código correspondiente a este primer programa:

```
1  #include <stdio.h>
2
3  int main()
4  {
5    printf("Hola Programación, Algoritmos y Estructuras de Datos");
6    return 0;
7  }
```

La línea 1 `(#include <stdio.h>)` es una directiva para el preprocesador que no forma parte del código ejecutable, e indica al mismo la importación de la declaración de las funciones a utilizar que están incluidas en los archivos de cabecera (*.h). En este caso, en el archivo de cabecera stdio.h se encuentran las declaraciones de todas las funciones de entrada/salida.

Las líneas 3 a 7 conforman la **función main()**, que es el programa principal:

- Esta sección de código debe existir en todo programa.
- Está formado por el encabezado `int main()`
- y el código ejecutable encerrado entre llaves {......}
- el código se ejecutará ordenadamente desde la llave superior { (inicio del algoritmo) hasta la llave inferior } (fin del algoritmo)

Las líneas 5 a 6 conforman el **código ejecutable**:

- Las sentencias terminan en ;
- Contienen expresiones, operaciones y llamadas a funciones.
- En los programas que desarrollemos a futuro también habrá:
  - estructuras de control
  - declaración de variables

La línea 5 es una llamada a la función `printf()`

- Esta función equivale a la instrucción “Escribir” en pseudocódigo o DF
- Solo puede utilizarse si se incluye stdio.h

La línea 6 es una instrucción de retorno:

- su ejecución finaliza con el programa
- debe ir al final del programa principal, antes de la llave de cierre (})

Identificadores – Variables

Las “variables” son los “datos” que hemos venido utilizando en el diseño de algoritmos en pseudocódigo, es decir, son nuestro modelo para un determinado dato que interviene en el algoritmo y que durante la ejecución del mismo tomará distintos valores. Es así, que a la hora de diseñar nuestros algoritmos, una de las primeras tareas es identificar nuestras variables y asignarles un tipo (número entero, número real, carácter, etc).

Los “identificadores” son las palabras creadas por el programador para dar nombre a las variables, constantes, tipos, funciones, etc.

- Se construyen con secuencias de una o más letras, dígitos o el símbolo _.
- Debe comenzar por una letra o símbolo de subrayado.
- El compilador reconoce los 32 primeros caracteres.
- El compilador de C es sensible a mayúsculas.
- No pueden tener la misma secuencia de caracteres que una palabra clave o una función de librería.

Ejemplos de identificadores válidos son: Dato1, dato1, Dato_1, _dato1, entre otros.

Ejemplo de identificadores NO válidos: 1Dato (porque empieza con un número), dato 1 (porque utiliza el espacio), etc.

La declaración de una variable en C tiene la siguiente expresión general:

```
<Modificadores de Tipo> <Tipo> <Nombre_Variable>;
```

donde:

- `<Nombre_Variable>:` debe ser un identificador válido.
- `<Tipo>:` char (para caracter), int (para entero), long (para entero largo), float (para real), double (real largo).
- `<Modificadores de Tipo>:` unsigned, signed, long, short.

Ejemplos:

```
char letra;
unsigned int valor;
float temperatura;
```

Tipos de datos básicos

Los tipos de datos básicos son:

- `char`: para caracteres o enteros de 8 bits.
- `short`: para enteros cortos (2 bytes)
- `int`: para enteros (2 o 4 bytes)
- `long`: para enteros largos (4 bytes)
- `float`: para reales, punto flotante simple precisión (4 bytes)
- `double`: para reales grandes, punto flotante doble precisión (8 bytes)

Se puede utilizar el modificador `unsigned` en la declaración de char, short, int y long para declararlo como variable **entera sin signo**.

El rango de representación para cada uno de los tipos es:

char	-128 a 127
unsigned char	0 a 255
short int	-32.768 a 32.757
unsigned short int	0 a 65.535
int	-2.147.438.648 a 2.147.483.647
unsigned int	0 a 4.294.967.295
long int	-2.147.438.648 a 2.147.483.647
unsigned long int	0 a 4.294.967.295
float	1,17x10exp-38 a 3,4x10exp38
double	2,22x10exp-308 a 1,79x10exp308

Expresiones – Asignación

En lenguaje C existen **expresiones de asignación** y, para escribirlas, se utilizan los siguientes operadores de asignación:

Operadores de Asignación:	
=	Asignación
+=	Suma y asignación
-=	Resta y asignación
*=	Producto y asignación
/=	División y asignación
%=	Módulo y asignación

De todos ellos, el más utilizado es el operador de asignación (=) y su funcionalidad es igual a la de una instrucción de asignación en pseudocódigo, siendo su sintaxis muy similar:

```
<Nombre_Variable>=<expresión>;
```

Existen dos diferencias:

1. En vez del símbolo flecha izquierda (←), se utiliza el carácter igual (=).
2. Se debe escribir un punto y coma (;) al final.

La `<expresión>` puede ser un valor o bien una operación aritmética, lógica o relacional. Por ejemplo:

```
temperatura =25,4;
temperatura=1,8*temperatura+32;
```

Cuando el operador de asignación (=) se escribe precedido de un operador aritmético: suma (+), resta (-), multiplicación (*), división (/) o módulo (%), la unión de ambos se convierte en un nuevo operador de asignación que opera de la siguiente manera:

"A la variable se le asigna el valor que se obtiene de evaluar `<variable> <operador_aritmético> <expresión>`"

Por ejemplo:

`resultado*=10;` produce el mismo resultado que `resultado=resultado*10;` pero se compila más rápido dado que resultado se evalúa sólo una vez en la primer expresión y dos veces en la segunda.

Operaciones aritméticas, relacionales y lógicas

En lenguaje C existen distintos tipos de operadores, que pueden ser aritméticos, lógicos o relacionales.

Operadores Aritméticos:	
+	Suma
-	Resta
*	Producto
/	División
%	Resto de la división entera
++	Incremento
--	Decremento

En el código que se muestra a continuación, se presenta un ejemplo de operadores aritméticos donde se utilizan la división entre reales, división entre enteros y el resto.

```
1   #include <stdio.h>
2   #include <stdlib.h>
3   //División real, entera y resto
4   int main()
5   {
6      int resultado, resto, suma = 22;
7      long contador = 7;
8      float resultado_real=suma/5.0;    /*resultado_real toma el valor
4.4*/
9        resultado = suma / contador;    /*resultado toma el valor 3*/
10     resto = suma % contador;          /*resto toma el valor 1*/
11     return 0;
12  }
```

En la línea 8 (`float resultado_real=suma/5.0;`)se realiza una operación de división entre reales, dado que uno de los operandos (denominador) es un real. En este caso, la variable resultado_real toma el valor 4.4.

En la línea 9 (`resultado = suma / contador;`)se realiza una operación de división entre enteros, tomando la variable resultado el valor 3.

En la línea 10 (`resto = suma % contador;`) se realiza una operación de resto de división entera, tomando la variable resto el valor 1.

En el código que se muestra a continuación, se presenta un ejemplo con los operadores aritméticos de incremento y decremento. En la primera parte (líneas 8-10) se utiliza el operador

unario de preincremento (notar que ++ está a la izquierda de x). En este sentido, la variable x se incrementa en 1 (pasa de valer 20 a 21) antes de asignarle su valor a la variable y. Como resultado de la operación, ambas variables (x e y) toman el valor 21.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  //Operadores aritméticos:
4  //Incremento y Decremento
5  int main()
6  {
7     int x,y;
8     //Preincremento
9     x = 20;
10    y = ++x; /* x = y = 21 */
11    //Posincremento
12    x = 20;
13    y = x++; /* y = 20, x = 21 */
14    return 0;
15 }
```

En la segunda parte (líneas 11-13) se utiliza el operador unario de posincremento (notar que ++ está a la derecha de x). En este sentido, la variable x se incrementa en 1 (pasa de valer 20 a 21) después de asignarle su valor a la variable y. Como resultado de esta operación, la variable y toma el valor 20, mientras que x toma el valor 21.

Operadores Relacionales:	
>	Mayor que
<	Menor que
>=	Mayor o igual que
<=	Menor o igual que
==	igual que
!=	distinto que

Los operadores relacionales se utilizan típicamente dentro de las estructuras de programación para evaluar condiciones y ejecutar una acción selectiva o bien repetitiva dependiendo del resultado de la operación. Este tipo de operadores trabajan sobre cualquier tipo de operandos y producen una salida binaria (Verdadero o falso / Uno o cero).

Por ejemplo:

```
#include <stdio.h>
#include <stdlib.h>
//Operadores relacionales:
//Igual que y distinto que
```

```c
int main()
{
int p, q;
float x = 15, y = 18;
p = (x==y); /*p toma el valor 0*/
q = (x!=y); /*q toma el valor 1*/
return 0;
}
```

Operadores Lógicos:	
&&	AND - Y
||	OR - O
!	NOT - NO

Ejemplo con operadores lógicos:

```c
#include <stdio.h>

int main()
{
int p, q, r;
float x = 15, y = 18, z = 20;
p = !(x == y);              /*p = 1*/
q = (x < y) && (y <= z)     /*q = 1*/
r = !x                      /*r = 0*/
}
```

Operadores a nivel Bit:	
&	AND - Y
|	OR - O
^	XOR
~	NOT (Complemento a 1)
>>	Desplazamiento a derecha
<<	Desplazamiento a izquierda

Ejemplo con operadores a nivel de bits:

```c
#include <stdio.h>

int main()
{
int a, m;
a = 2;
```

```
b =5;
m= 7;
a = a | m;    /*a=7*/
b = b & m;    /*b=5*/
b = a << 1;   /*b=14*/
}
```

En la Tabla 3.1 (extraída de http://lsi.vc.ehu.es/asignaturas/FdIc/labs/a1/htm/oper.html) se incluyen todos los operadores antes descriptos ordenados por nivel de precedencia:

Tabla 3.1. Precedencia de Operadores en C.

Nivel	Operadores	Descripción	Asoci.
1	`() [] -> .`	Acceso a un elemento de un vector y paréntesis	Izquierdas
2	`+ - ! ~` `* &` `++ --` `(cast) sizeof`	Signo (unario), negación lógica, negación bit a bit Acceso a un elemento (unarios): puntero y dirección Incremento y decremento (pre y post) Conversión de tipo (*casting*) y tamaño de un elemento	**Derechas**
3	`* / %`	Producto, división, módulo (resto)	Izquierdas
4	`+ -`	Suma y resta	Izquierdas
5	`>> <<`	Desplazamientos	Izquierdas
6	`< <= >= >`	Comparaciones de superioridad e inferioridad	Izquierdas
7	`== !=`	Comparaciones de igualdad	Izquierdas
8	`&`	Y (*And*) bit a bit (binario)	Izquierdas
9	`^`	O-exclusivo (*Exclusive-Or*) (binario)	Izquierdas
10	`|`	O (*Or*) bit a bit (binario)	Izquierdas
11	`&&`	Y (*And*) lógico	Izquierdas
12	`||`	O (*Or*) lógico	Izquierdas
13	`?:`	Condicional	**Derechas**
14	`= *= /= %= += -= >>=` `<<= &= ^= |=`	Asignaciones	**Derechas**
15	`,`	Coma	Izquierdas

Entrada/salida de datos

Funciones para entrada de un carácter desde teclado:

- Función getchar:

```
prototipo:   int getchar(void);
```

Presente en la librería stdio.h. Esta función guarda los caracteres que se tecleen en el buffer de entrada, hasta que se pulse <ENTER>. Después recoge sólo el primer carácter y lo almacena

en la variable especificada, pudiendo quedar uno o más caracteres en el buffer de entrada. Así, las posteriores funciones de entrada en el programa leerán esos caracteres del buffer, obteniéndose resultados inesperados. Se recomienda el uso de la función fflush(stdin) para vaciar el buffer después de su uso.

Ejemplo de uso:

```
char letra;
letra = getchar();
```

En caso de error devuelve EOF, constante definida en stdio.h con valor igual a –1.

- Función getche:

prototipo: `int getche(void);`

Está en la librería conio.h. El carácter tecleado será almacenado y visualizado en pantalla.

Ejemplo de uso:

```
char letra;
letra = getche();
```

- Función getch:

prototipo: `int getch(void);`

Está en la librería conio.h. Funciona exactamente como la anterior, excepto que el carácter tecleado no será visualizado en pantalla (sin eco).

Función de entrada con formato:

- Función scanf:

prototipo: `int scanf ( char * cadena_control, lista_de_argumentos);`

La lista_de_argumentos está formada por las direcciones de memoria (punteros) de las variables donde se quiere guardar los valores tecleados. Por tanto, debe usarse el operador & delante del nombre de cada variable.

Cada especificador de formato se corresponderá con una variable de la lista de argumentos. Está en la librería stdio.h, en caso de error devuelve EOF.

Ejemplo de uso:

```
int num, dato;
char car;
scanf("%d %c %d", &num, &car, &dato);
```

Código	Significado
% c	Lee un único carácter
% d	Lee un entero
% i	Lee un entero
% ld	Lee un entero de tipo *long*
% e	Lee un número en coma flotante
% f	Lee un número en coma flotante
% lf	Lee un número en coma flotante *double*
%Lf	Lee un número en coma flotante *long double*
% g	Lee un número en coma flotante
% o	Lee un número octal
% s	Lee una cadena
% x	Lee un número hexadecimal
% p	Lee un puntero
% n	Recibe un valor entero igual al número de caracteres leídos
% u	Lee un entero sin signo

```
%[nº_de_caracteres]type
```

El nº de caracteres será un número entero opcional que se coloca entre % y type. Ese número indica el número máximo de caracteres a almacenar en la variable correspondiente, de forma que el exceso no se tendrá en cuenta.

Ejemplos de uso:

```
int edad;
scanf( "%3d",&edad);   /*Después de teclear 3 números, lo que se teclee no
se guarda.*/

int i,j;
scanf( "%o %x", &i, &j);     /*Uno en octal y otro en hexa.*/

char a,b,c;
scanf( "%c %c %c", &a, &b, &c);       /* Lee tres letras.*/
```

Salida de datos

Funciones para sacar un carácter por pantalla:

- Función putchar:

prototipo: `int putchar(int c);`

Está en la librería stdio.h. El entero c es convertido a carácter y se envía al monitor. En caso de error devuelve EOF.

Ejemplo:

```
char car;
car = getchar();
putchar(car);
```

Función de salida con formato:

- Función printf:

prototipo: `int printf(char * cadena_control, lista_de_argumentos);`

Esta función puede visualizar varios datos por pantalla. printf: está en la librería stdio.h. Devuelve un valor entero que coincide con el número de caracteres visualizados. En caso de error, devuelve un valor negativo.

La cadena_control se escribe entre comillas dobles y está formada por especificadores de formato y por los caracteres que se desee visualizar.

Los especificadores de formato tienen la siguiente forma:

%[flags] [ancho de campo] [precisión] type

% y type son caracteres obligatorios, el resto opcionales (entre corchetes).

La lista_de_argumentos del prototipo de printf está formada por las variables o constantes que se quiera visualizar. Cada argumento de la lista debe tener su especificador de formato.

El modificador type del especificador de formato puede ser:

CÓDIGO	**FORMATO**
%c	Carácter
%d	Enteros con signo
%i	Enteros con signo
%ld	Enteros con signo long
%e	Coma flotante, notación científica (e minúscula)
%E	Coma flotante, notación científica (E mayúscula)
%f	Coma flotante (float)
%lf	Coma flotante (double)
%Lf	Coma flotante (long double)
%g	usa e o f, el más corto
%G	usa E o f, el más corto
%o	Octal sin signo
%s	cadena de caracteres
%u	entero sin signo
%x	Hexadecimal sin signo (a-f)
%X	Hexadecimal sin signo (A-F)
%p	Muestra un puntero

ancho de campo : es un número entero que especifica la longitud mínima que ocupará el dato al visualizarse, teniendo en cuenta el punto decimal y los decimales.

Ejemplo:

```
int i = 100;
printf("i = %6d", i); /* Sale i = bbb100 */
```

Si la longitud del dato es mayor que la especificada en el formato, se visualiza todo el dato, no se tiene en cuenta esa longitud mínima. Si en lugar de rellenar con blancos, se desea rellenar con ceros, se coloca un 0 entre el % y el entero que indica la longitud:

Ejemplo:

```
int i = 100;
printf( "i = %05d", i); / *Sale i = 00100 */
```

precisión: se coloca entre el % y el type, después del ancho del campo con la forma .número. Indica la precisión, el número de decimales a visualizar, completando con ceros por la derecha si es necesario, o truncando los decimales si no caben en la precisión indicada (redondeando al valor más cercano).

Ejemplo:

```
float num = 100.18;
printf("Número = %f", num); /*Sale: f = 100.180000*/
printf("Número = %7.1f",num); /*Sale: f = bbbb100.2*/
printf("Número = %.3f", num); /*Sale: f = 100.180*/
```

Si se usa con cadenas (%s), la precisión indicará la longitud máxima que se visualizará; la longitud mínima viene dada por el ancho de campo indicado.

“%8.10s” Escribe la cadena de 8 a 10 caracteres.

Flag:

-	justifica a izquierda, inserta blancos a derecha
+	Visualiza el signo +/-
b	Un blanco hace que sólo visualice el signo negativo, y en lugar del positivo sale un blanco

Ejemplo:

```
printf ("|%-15s|", "hola"); /*Sale |holabbbbbbbbbbb|*/
printf ("|%15s|", "hola"); /*Sale |bbbbbbbbbbbhola|*/
```

Existen las secuencias de backslash (o secuencias de escape) para realizar operaciones de control que pueden ser ejecutadas con printf.

Secuencia	Significado
\a	Sonido
\b	Espacio atrás (backspace)
\f	Salto de página (sólo impresora)
\n	Salto de línea
\r	Retorno de carro
\t	Tabulación horizontal
\v	Tabulación vertical (sólo impresoras)
\\	Barra invertida
\'	Comilla simple
\"	Comilla doble
\0	Nulo
\ddd	Tres dígitos que son la rep.octal del carácter ASCII
\xdd	Una 'x' más dos dígitos que son la rep.hexadec. del carácter ASCII.

Estructuras de control

A continuación se presenta la implementación de las distintas estructuras de control en lenguaje C, en base a ejemplos sencillos, presentando los algoritmos con diagrama de flujo y su equivalente en C.

Selectiva Simple:

```
#include<stdio.h>
//Estructura Selectiva Simple
main()
{
/*Declaración de variables*/
float NOTA;
/* leer nota de alumno*/
printf("Ingrese nota: ");
scanf("%f", &NOTA);
if (NOTA < 4.0)
  printf("Desaprobado. \n");
}
```

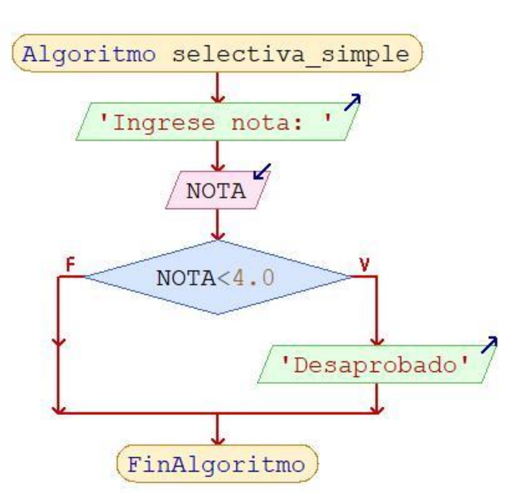
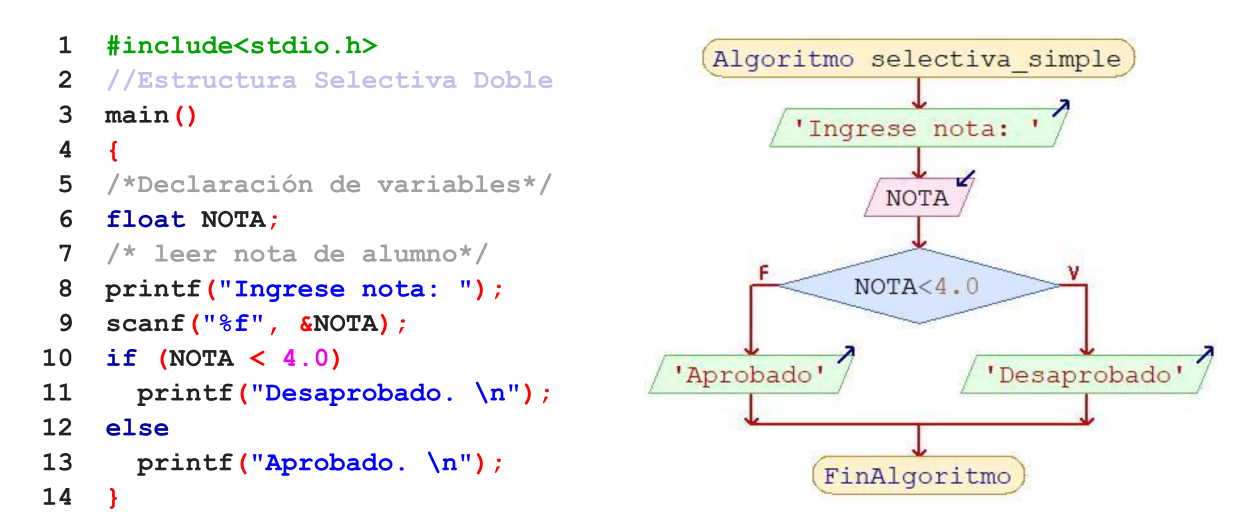


Figura 3.7. *Selectiva Simple.*

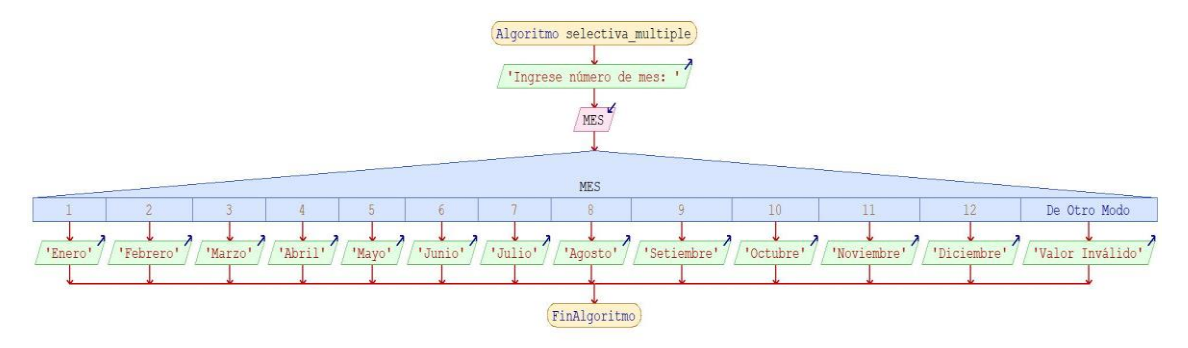
Selectiva Doble:



```c
#include<stdio.h>
//Estructura Selectiva Doble
main()
{
/*Declaración de variables*/
float NOTA;
/* leer nota de alumno*/
printf("Ingrese nota: ");
scanf("%f", &NOTA);
if (NOTA < 4.0)
  printf("Desaprobado. \n");
else
  printf("Aprobado. \n");
}
```


Figura 3.8. *Selectiva Doble*

Selectiva Múltiple:

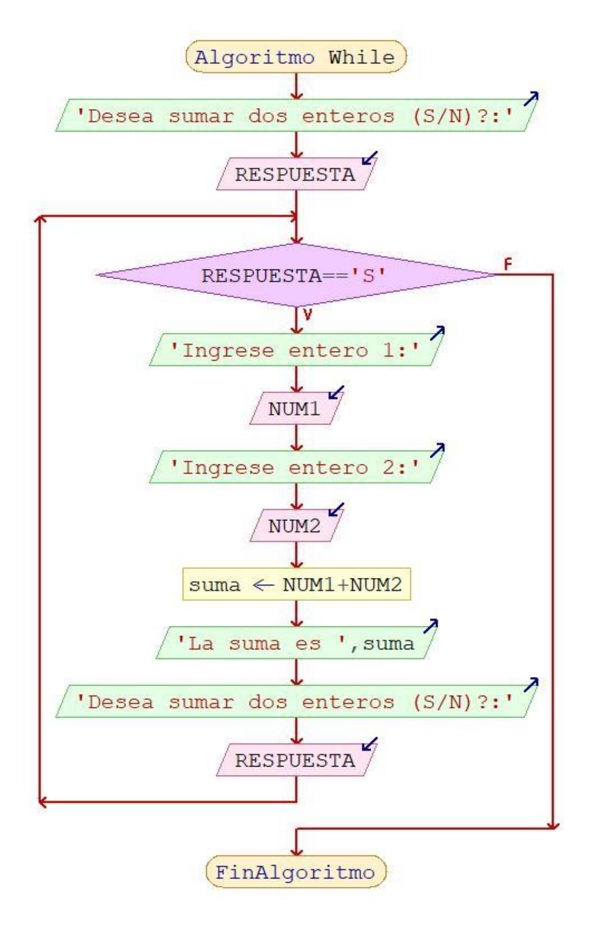



```c
#include<stdio.h>
//Estructura Selectiva Múltiple
main()
{
/*Declaración de variables*/
int MES;
/* leer número de mes*/
printf("Ingrese numero de mes: ");
scanf("%d", &MES);
switch (MES)
  {
  case 1:
        printf("Enero\n");
        break;
  case 2:
        printf("Febrero\n");
        break;
  case 3:
        printf("Marzo\n");
        break;
  case 4:
        printf("Abril\n");
        break;
  case 5:
        printf("Mayo\n");
        break;
```

```
        case 6:
              printf("Junio\n");
              break;
        case 7:
              printf("Julio\n");
              break;
        case 8:
              printf("Agosto\n");
              break;
        case 9:
              printf("Setiembre\n");
              break;
        case 10:
              printf("Octubre\n");
              break;
        case 11:
              printf("Noviembre\n");
              break;
        case 12:
              printf("Diciembre\n");
              break;
        default:
              printf("Valor Incorrecto\n");
        }
    }
```

Figura 3.9. *Selectiva Múltiple*

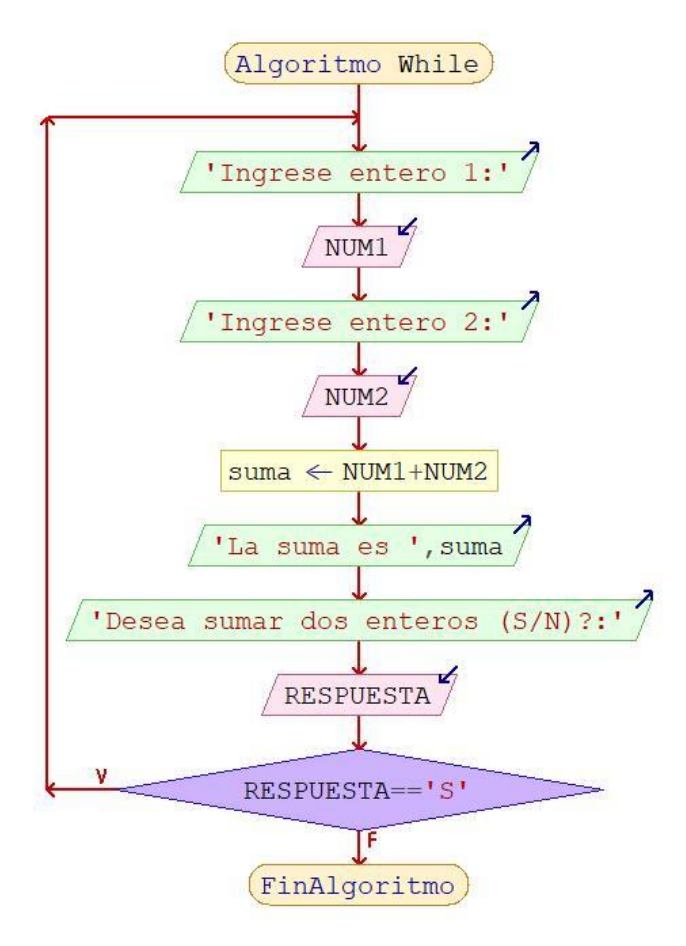
Repetitiva While/Mientras:



```
#include<stdio.h>
//Estructura Repetitiva While
int main()
{
int NUM1, NUM2, suma;
char RESPUESTA;
printf("Desea sumar dos enteros (S/N)?: ");
scanf(" %c", &RESPUESTA);
while (RESPUESTA=='S')
    {
    printf("Ingrese entero 1: ");
    scanf(" %d", &NUM1);
    printf("Ingrese entero 2: ");
    scanf(" %d", &NUM2);
    suma = NUM1 + NUM2;
    printf("La suma es %d \n", suma);
    printf("Desea sumar dos enteros (S/N)?: ");
    scanf(" %c", &RESPUESTA);
    }
 return 0;
 }
```

Figura 3.10. *Estructura repetitiva while*

Repetitiva do-While/Hacer-Mientras:

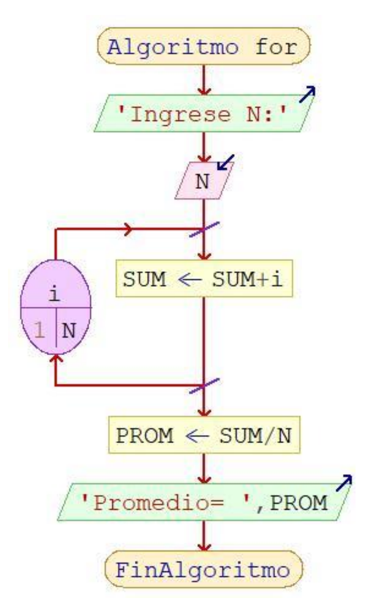


```c
#include<stdio.h>
//Estructura Repetitiva do-While
int main()
{
int NUM1, NUM2, suma;
char RESPUESTA;

do
  {
  printf("Ingrese entero 1: ");
  scanf(" %d", &NUM1);
  printf("Ingrese entero 2: ");
  scanf(" %d", &NUM2);
  suma = NUM1 + NUM2;
  printf("La suma es %d \n", suma);
  printf("Desea sumar dos enteros (S/N)?: ");
  scanf(" %c", &RESPUESTA);
  }while (RESPUESTA=='S');
return 0;
}
```

Figura 3.11. *Estructura repetitiva do- while*

Repetitiva for/para:




```c
#include<stdio.h>
int main()
{
int N, i, SUM = 0;
float PROM;
printf("Ingrese N: ");
scanf("%d",&N);
for (i=1;i<=N;i++)
  {
  SUM = SUM + i;
  }
PROM = (float)SUM/(float)N;
printf("Promedio = %f \n", PROM);
return 0;
}
```

Figura 3.12. *Estructura repetitiva for*

Ejemplo integrador

Enunciado: Haciendo uso de las funciones de entrada/salida y las estructuras de control, implemente el código en C de un algoritmo que recibe números enteros, y contabiliza mostrando en pantalla, la cantidad de dígitos pares e impares presentes en el entero recibido. Deberá dar la posibilidad al usuario del programa de ingresar tantos enteros como desee, proponer una forma para finalizar el ingreso de datos.

Solución:

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
 int num,pares=0,impares=0,resto;
 char respuesta='s';
 while(respuesta=='s' || respuesta=='S')
 {
 printf("Ingrese un entero:");
 scanf("%d",&num);
 fflush(stdin);
 while(num!=0)
        {
        resto=num%10;
        if(resto%2==0)
        pares++;
        else
        impares++;
        num/=10;
        }
 printf("\nEl # tiene %d digitos pares y %d impares\n",pares, im-
pares);
 pares=impares=0;
 printf("Desea ingresar otro entero (s/n)");
 scanf("%c",&respuesta);
 }

 return 0;
 }
```

Ejercicios

Programación Estructurada en Lenguaje C

1) Familiarícese con el ambiente de trabajo (IDE) Code::Blocks y el proceso de codificación, compilación y ejecución de un programa:

a) Cree un proyecto nuevo, prestando atención a la carpeta donde ubica el proyecto. Tras crearlo, busque la carpeta del proyecto e identifique
   i. El archivo de código C con la plantilla creada por defecto
   ii. El archivo del proyecto de Code::Blocks
b) Compile el programa que el entorno crea por defecto y verifique que no haya errores ni advertencias. Verifique que en la carpeta de proyecto se hayan creado nuevas carpetas. ¿Dónde está el ejecutable?
c) Ejecute el programa desde el entorno, y luego desde el ejecutable que encontró en la carpeta. ¿Qué diferencias observa?

Nota: Puede utilizar el apunte “Tutorial Code::Blocks” que encontrará en la página de la cátedra para ayudarse.

2) Se tiene el siguiente fragmento de código:

```
char a = 120;  /* Se declara la variable “a” y se inicializa */
printf(“a: %d”, a); /* Se imprime en pantalla el valor de a */
```

a) Prográmelo y ejecútelo en el IDE para corroborar que hace lo que espera.
b) Cambie el valor de inicialización de `a` por 150. ¿Qué sucede?
c) Busque cuál es el máximo valor positivo que puede almacenar en a. Puede utilizar este programa para “ir probando” o puede consultar la teoría
d) Cambie el tipo de `a`, en lugar de char pruebe con unsigned char y repita los casos anteriores para 120, 150 y 260. Comente lo observado.

3) Indique con que tipo(s) de dato(s) (`char`, `short`, `int`, `float`, `double`, etc.) declararía variables para almacenar cada uno de los siguientes valores:
a) 3.1416
b) 5000
c) '3'
d) 3

4) Realice el código que
a) Declare las variables del punto anterior
b) Les asigne los valores correspondientes
c) Imprima en pantalla dichos valores en el formato adecuado
d) Repita el ejercicio, pero en el inciso b) permita que el usuario ingrese el valor de cada variable por teclado

5) Realice el programa que luego de ejecutarse se verá como sigue:

```
***    Calculadora de panqueques     ***
*** Encante a sus invitados con unos ***
*** buenos panqueques                ***

¿Cuántos invitados tiene? (1-100)
Ingrese el número: 6

¿Cuantos panqueques comerá cada invitado? (1-10)
Ingrese el número: 3
La receta para 6 invitados y 3 panqueques por invitado es:

        Ingrediente     cantidad        unidad
        --------------------------------------
        Huevos          3,0             unidades
        Leche           500,0           mililitros
        Harina          250,0           gramos
        Sal             1,0             cucharaditas
        Vainilla        1,0             chorritos


Que disfrute los panqueques!
```

Nota: Para imprimir la tabla puede usar la secuencia de escape `\t` en el `printf`

6) Analice qué se imprimirá en pantalla al ejecutar los siguientes fragmentos de código, luego ejecútelos y compruebe sus predicciones. Nota: Si copia y pega el código desde el PDF, esté atento a posibles errores de copiado que pueden impedir la compilación.

a)

```
int metros;
float kilometros;
printf("Ingrese una distancia en metros para convertirla: ");
scanf("%d",&metros);
kilometros = metros/1000;
printf("Resultado: %f",kilometros);
```

b)

```
int a;
a=0;
printf("%d\n",a++);
printf("%d\n",++a);
a++;
printf("%d\n",++a);
```

c)

```
char x;
x = 'h';
printf("%c %d\n",x,x);
x = 105;
printf("%c %d",x,x);
```

7) Codifique en lenguaje C los siguientes algoritmos planteados en la práctica Nº 1:

- Algoritmos 1, 2, y 5 del ejercicio 9
- Ejercicios 14, 15, 18 y 19

Puede valerse de los enlaces útiles de la página de la cátedra u otra referencia de programación en C.

8) Encuentre los errores en los códigos disponibles en la página de la cátedra (en la carpeta Descargas/CodigosP2) y corríjalos para asegurar que los programas compilen y se ejecuten apropiadamente.

CAPÍTULO 4
Tipos de datos básicos y arreglos

Leandro Mendez, Alejandro Moyano y Juan M. Rosso

En este Capítulo se presentan nociones vinculadas con tipos de datos, como concepto que está en la base de la representación de datos en el lenguaje de programación.

En su forma más básica, los datos forman, en el interior de la computadora, un conjunto de bits. A partir de ellos, el diseñador de un lenguaje puede desarrollar datos. El lenguaje de programación utilizado incluye un conjunto de entidades tales como enteros, reales, etc. y mecanismos para obtener nuevas entidades a partir de éstas (entre otros, los llamados arreglos, como se verá más adelante).

Las dependencias de la máquina, son parte de la implementación de los tipos de datos. Como ejemplo de esto, puede mencionarse el carácter finito de todos los datos en una computadora. Es así que al hablar de un dato entero, podríamos pensar, en sentido matemático, como el conjunto Z de los números enteros…,-2,-1,0,1, 2,…; pero en una computadora siempre existe un mínimo y un máximo entero posible de representar.

Una situación similar surge con la precisión de los números reales y el comportamiento de las operaciones aritméticas con ellos. A fin de reducir la dependencia del hardware de las operaciones con números reales, se estableció a partir de 1985, el standard IEEE 754 de representación de punto flotante.

Los tipos de variables a menudo se asocian explícitamente con las variables por una declaración tal como:

```
int x;
```

lo cual asigna el tipo de dato `int` a la variable `x`. En esta declaración la palabra reservada int lleva consigo cierta información, tal como los valores que pueden almacenarse y la forma en que se representan internamente.

En Louden 2011, se define un tipo de dato como un conjunto de valores con un conjunto de operaciones sobre esos valores.

Es así que la declaración de x como int, dice que el valor de `x` debe estar en el conjunto de los enteros definidos por el lenguaje y la implementación.

Las operaciones son parte de la definición del tipo. Por ejemplo, se cuenta con las operaciones aritméticas sobre los enteros o los reales.

Un lenguaje permite hacer uso correcto de datos y operaciones en un programa. Por ejemplo, en C, si `x e y` son de tipo `int` (con `y` distinto de cero), entonces

```
z = x / y
```

significa una división entera y el resultado de ésta también es `int.`

Determinar si el tipo de información en un programa es consistente se llama chequeo de tipo. En el ejemplo precedente, se verifica que las variables `z, x, e y` son usadas correctamente en esa asignación.

Dado un conjunto de tipos básicos como `int, float, double y char`, el lenguaje ofrece varias posibilidades de construir tipos más complejos fuera de los básicos. En este sentido, por ejemplo, el arreglo (array), cuya presentación se incluye en párrafos posteriores, toma un tipo base (digamos `int` para fijar ideas) y un tamaño para ese arreglo.

Como se mencionó, el lenguaje cuenta con un conjunto de tipos predefinidos, especificados usando palabras clave, tales como `int o double`, a partir de los cuales son construidos todos los otros tipos. También se tienen predefinidas variaciones tales como `unsigned, short`, etc de los tipos básicos.

En este punto es oportuno destacar que el lenguaje C ofrece la posibilidad de reserva de memoria para el almacenamiento temporal de una variable y su uso en el programa. Tal como su nombre lo indica, el contenido de una variable podría ser modificado durante la ejecución del programa. Tales variables se alojarán en un área del programa, conocida como segmento de datos. Contrariamente, aquellas declaraciones de memoria de tipo constante o con acceso de solo lectura, se alojarán en el segmento de código.

Utilización de los tipos de datos

Como ya se mencionó, se puede hacer uso de cada uno de los tipos de datos, con una declaración, es decir especificando el tipo a través de las siguientes palabras reservadas: `char, int, float, double, void` y luego asignando un nombre a la variable.

La siguiente línea de código, ilustra la sintaxis de una declaración, para la reserva de memoria para una variable de tipo carácter:

```
char aux;          /* declaración de variable aux de tipo char */
```

El requerimiento de memoria, está relacionado con el tamaño en bits disponible según diseño, siendo por lo común para los tipos básicos `char, int, float, double`, de 8, 32, 32 y 64 respectivamente. Pudiendo presentarse otras variantes, dado que se trata de características directamente asociadas con la herramienta de compilación y la longitud de palabra del hardware para el cual se está compilando el código.

Con el uso del operador `sizeof`, puede determinarse el tamaño de la variable en cuestión.

Respecto de la palabra reservada `void`, se puede decir que, en el contexto de la programación, tiene más de un significado, con lo cual la traducción al español podría generar confusión. Se pospone su uso para posteriores capítulos.

Cada tipo de dato tiene asociado un rango de valores que una variable podría albergar, además de ello también se define el tipo de operaciones disponibles sobre dicha variable.

Operaciones con enteros

Operaciones algebraicas, suma, resta, multiplicación, división, módulo o resto aritmético. Operaciones relacionales y lógicas, operaciones lógicas a nivel de bit.

Pueden ser utilizadas como índice (ver arreglos), como condicional, particularmente útil en selectiva múltiple switch.

Los modificadores

Existe la posibilidad de alterar el rango de representación de datos en el caso de variables de tipo entero, si al momento de su declaración se hacen con signo o sin signo, (es decir si representará números enteros en complemento a dos o números enteros sin signo en binario puro, respectivamente). Tal opción se implementa en el lenguaje C, anteponiendo la palabra reservada `unsigned` al tipo de dato entero que desea modificarse.

Sin llegar a ser exhaustiva, a continuación se hace una presentación de algunos modificadores utilizados en el lenguaje.

También existe la posibilidad de alterar el ancho en bits, ampliando el rango de representación, mediante el uso de la palabra reservada `long`.

Como así también, haciendo uso de la palabra reservada `register`, se comunica al compilador la preferencia de que una variable de programa, sea alojada directamente en registros especiales del procesador, en lugar de hacerlo reservando memoria de acceso aleatorio como más frecuentemente se dá.

Su propósito es el de reducir el tiempo de acceso a un dato, dado que no resulta necesario acceder a un bus de datos para comunicarse con el dispositivo de almacenamiento de memoria. Como desventaja puede mencionarse que sólo es posible reservar pequeñas cantidades de memoria, del orden de algunos bytes.

El código que se muestra a continuación ilustra el uso de los modificadores `unsigned y long` para el almacenamiento de números enteros sin signo en una variable de 64 bits. La variable `x` se inicializa con el número más grande que se puede representar ($2^{64} - 1$), poniendo todos sus bits en 1, utilizando base hexadecimal para una representación más compacta. El número en cuestión tiene veinte cifras significativas en base decimal.

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
	unsigned long long int x = 0xFFFFFFFFFFFFFFFF;
	printf("%I64u\n", x);
	return 0;
}
```

Mediante el uso del modificador `const` se restringe el acceso a los datos, estableciendo permisos de solo lectura, con la finalidad de impedir la modificación involuntaria de su contenido (ver código a continuación).

De este modo, una línea de código que tuviese por finalidad modificar los datos, provocaría un error en tiempo de compilación, obligando al programador a rectificarlo (Tabla 4.1).

```
 1  #include <stdio.h>
 2  #include <stdlib.h>
 3
 4  int main()
 5  {
 6          const char texto[] ="Tipos de datos";
 7          texto[0] = 't';
 8          printf("%s\n", texto);
 9          return 0;
10  }
```

Tabla 4.1. Mensaje de error

File	Line	Message
C:\main.c	7	error: assignment of read-only location 'texto[0]'
		=== Build failed: 1 error(s), 0 warning(s) (0 minute(s), 0 second(s)) ===

La siguiente tabla 4.2 resume algunas de las características, mencionadas anteriormente:

Tabla 4.2. Características de los tipos de datos

Tipo	Descripción	Requerimiento típico de memoria	Rango
Char	Tipo de carácter puede almacenar códigos ASCII	8 bits	-128 a 127
unsigned char	Almacena números enteros sin signo	8 bits	0 a 255
int	Tipo de entero, almacena números enteros en complemento a dos	32 bits	-2.147.483.648 a +2.147.438.647
unsigned int	Almacena números enteros en sin signo	32 bits	0 a + 4.294.967.295
unsigned long int	Almacena números enteros en sin signo	64 bits	18.446.744.073.709.551.615

<table>
<tr>
<td>float</td>
<td>Almacena números de coma flotante, hasta seis cifras significativas.
<table>
<tr><td>Signo</td><td>1 bit</td></tr>
<tr><td>Exponente</td><td>8 bits</td></tr>
<tr><td>mantisa</td><td>23 bits</td></tr>
</table>
</td>
<td>32 bits</td>
<td>Min. @ ±1,18 • 10^{-38}
Max. @ ±3,40 • 10^{+38}
Representaciones de ±inf, ±0 y NaN</td>
</tr>
<tr>
<td>double</td>
<td>Almacena números en coma flotante en precisión extendida
<table>
<tr><td>Signo</td><td>1 bit</td></tr>
<tr><td>Exponente</td><td>11 bits</td></tr>
<tr><td>mantisa</td><td>52 bits</td></tr>
</table>
</td>
<td>64 bits</td>
<td>Min. @ ±1 • 10^{-308}
Max. @ ±1 • 10^{+308}
Representaciones de ±inf, ±0 y NaN</td>
</tr>
</table>

Arreglos

Los arreglos son colecciones de datos de un mismo tipo, asociados a un único nombre o etiqueta, los cuales se almacenan en la memoria del ordenador de manera consecutiva. El primer elemento del arreglo siempre corresponde a la posición de memoria de menor valor, y la misma está asociada directamente al nombre del arreglo (esta propiedad se profundizará en el capítulo de punteros). Los arreglos deben tener tamaño o dimensiones predefinidos excepto para los arreglos indeterminados.

Existen varios tipos de arreglos de datos:

- Unidimensionales (vectores)
- Bidimensionales (matrices)
- Multidimensionales (3 o más dimensiones)
- Cadenas de caracteres
- Arreglo de cadenas de caracteres
- Arreglos indeterminados (sin tamaño predefinido)

Arreglos unidimensionales

Los arreglos unidimensionales contienen un número definido de elementos de un mismo tipo de datos, accesibles mediante un único índice.

Definición

```
Tipo_Dato Nombre_Arreglo [Tamaño];
```

`Tipo_Dato:` es el tipo de datos que almacena el arreglo, puede ser cualquiera de los tipos de datos ya conocidos como: `int, char, float, double,` etc.

`Nombre_Arreglo`: etiqueta definida por el usuario para referirse a la colección de datos o arreglo.

`Tamaño:` es un número natural mayor que cero el cual determina el tamaño o cantidad de elementos del arreglo.

Ejemplos:

```
int vector[4];                //arreglo de 4 enteros

float presiones[8];       //arreglos de 8 flotantes.
```

Acceso a los datos en arreglos

El acceso a los datos de un arreglo unidimensional se hace a través de la combinación del nombre del arreglo y un índice i.

```
Nombre_Arreglo [indice]
```

Como regla en el lenguaje C, para acceder al elemento i-esimo [i] se debe invocar al índice `[i-1]`. O sea que el primer elemento de un vector tiene siempre índice 0 (i=0). Es por esto, que, si el tamaño de un arreglo es N, i podrá tomar valores entre 0 y N-1.

Ejemplo:

```
vector[0];            //accede al primer elemento del arreglo vector
presiones[2];      //acceder al tercer elemento del arreglo presiones
```

Tip: el programador debe tener el control del tamaño de los arreglos y los valores máximos de los índices para acceder a sus elementos ya que el compilador no informa un error en caso de intentar acceder a un elemento de índice mayor que el tamaño del arreglo. En este caso, se accederá a un valor indeterminado que puede provocar el mal funcionamiento del algoritmo.

Carga de datos en arreglos

La carga de datos se puede hacer por inicialización en tiempo de compilación:

```
Tipo_Dato Nombre_Arreglo [N]={valor1,valor2,…….,valorN-1};
```

Ejemplo:

```
int vector[5] = {1,1,2,3,5};
float presiones[8] = {2.1, 3.5, 4.6, 4.7, 5.7, 7.8, 10.2, 1.0};
```

La cantidad de elementos cargados en el arreglo no debe superar el tamaño del mismo. Sí puede ser menor, y los restantes elementos se inicializarán en 0 o tomarán el valor preexistente en dicha posición de memoria dependiendo del compilador utilizado.

Otra posibilidad para la carga de datos es a través de la asignación.

Ejemplo:

```
int vector[3];
vector[0]=1;        //asigna el valor 1 al primer elemento del arreglo
vector[1]=3;        //asigna el valor 3 al segundo elemento del arreglo
vector[2]=5;        //asigna el valor 5 al tercer elemento del arreglo
```

Tip: El tamaño total en bytes del arreglo dependerá del tipo de datos que almacene. Para conocer este valor se debe utilizar el operador sizeof(tipo_dato) colocando como parámetro el nombre del vector del cual se quiera conocer el tamaño total.

El código 4.1 implementa un ejemplo integrador donde se define el arreglo, se presenta carga de datos por el usuario y presentación en pantalla de todos los elementos del arreglo.

```
#include <stdio.h>
#include <stdlib.h>
#include <stdio.h>
#define  TAM 3 /*constante que define el tamaño del arreglo*/

void main(void)
{
int elem[TAM], i;   /* definición del arreglo y variable índice*/
for(i=0; i<TAM; i++)  /*Bucle para carga de valores en el arreglo*/
   {
   printf ( "Introduzca valor con índice %d: ", i);
   scanf ("%d", &elem[i]);
   }
for(i=0; i<TAM; i++) /*Impresión en lista de elementos del arreglo*/
   printf("El valor %d tiene índice %d\n", elem[i], i );

```

Código 4.1. *Ejemplo integrador arreglos unidimensionales*

Tip: al momento de trabajar con arreglos es recomendable utilizar etiquetas mediante la sentencia #define para definir el tamaño de los arreglos ya que en caso de ser necesario modificar el tamaño del arreglo, solo bastará con modificar el valor de la etiqueta y no se deberá modificar cada línea de código que invoque o se refiera al tamaño del arreglo.

Arreglos bidimiensionales

Son arreglos de 2 dimensiones los cuales pueden ser asociados al concepto matemático de matriz las cuales distribuyen los datos en filas y columnas. Son útiles para representar datos asociados, por ejemplo la evolución de temperaturas a lo largo de los días de un mes. (Temp vs tiempo), notas en diferentes materias de un listado de alumnos (Nro alumno vs Notas).

Como en los arreglos unidimensionales todos los datos deben ser del mismo tipo.

Definición

```
Tipo_Dato Nombre_Arreglo [Tamaño1] [Tamaño2];
```

`Tamaño1:` número de filas del arreglo

`Tamaño2:` número de columnas del arreglo

En la figura 4.1 se representa un arreglo bidimensional de M filas x N columnas donde se puede observar el movimiento de los índices.

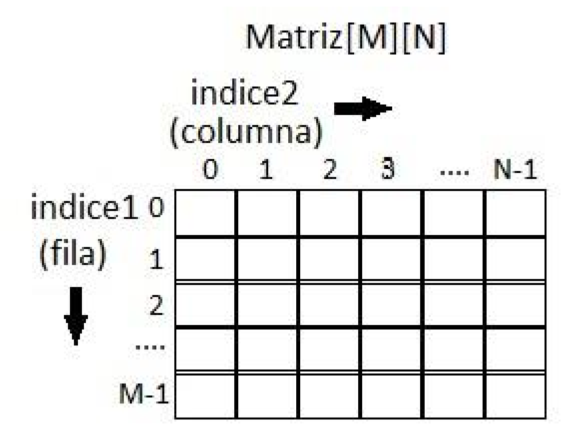


Figura 4.1. *Arreglo bidimensional*

Ejemplos:

```
int matrix[4][8];       //Matriz de enteros de 4 filas y 8 columnas
```

El acceso a los datos en arreglos bidimensionales se hace a través del nombre del arreglo y un par de índices:

```
Nombre_Arreglo [indice1] [indice2];
```

Ejemplo:

```
matrix[1][7];     //Accede al octavo elemento de la segunda fila
```

Carga de datos en arreglos bidimensionales

Por inicialización

`int matrix[2][4] = {1,2,3,4,5,6,7,8};` *//Todos los elementos en una sola instrucción*

Por Asignación

```
int matrix[0][0]=1;
int matrix[0][1]=2;
int matrix[0][2]=3;
int matrix[0][3]=4;
```

Tip: a pesar de utilizar 2 índices para acceder a un dato, todos los elementos de un arreglo se encuentran distribuidos de forma secuencial en posiciones contiguas de la memoria del ordenador.

En el código 4.2 se ejemplifica con el uso de un arreglo bidimensional de 3x3 enteros imprimiendo en pantalla los valores de la diagonal principal de la misma.

```
#include <stdio.h>

int main()
{
int matrix [3][3] = {1,2,3,4,5,6,7,8,9}; /*Inicializa la matriz*/
int i, j; /* Se declaran variables índice */
for (i=0;i<3;i++)  /*visualiza los elementos de la diagonal*/
{
    for (j=0;j<3;j++)
    {
     if(i==j)
      printf("%d\n", matrix[i][j]);//se imprime el valor de la diagonal
    }
}
return 0;
}
```

Código 4.2. *Ejemplo de uso arreglo bidimensional*

Arreglos multidimensionales

Es una generalización de los arreglos a N dimensiones. Por ejemplo, si queremos llevar un sobre a un determinado departamento ubicado en un condominio de edificios, deberemos saber número de unidad o edificio, el piso y finalmente número de departamento dentro del piso, por lo que necesitaremos 3 dimensiones para llegar al departamento correcto. Y por ejemplo en caso de querer ubicar una partícula en el espacio según Einstein necesitaremos 4 dimensiones.

Definición

Tipo_Dato *Nombre_Arreglo* [***Tamaño1***] [***Tamaño2***]**….[TamañoN];**

Acceso a los datos

Nombre_Arreglo [***indice1***] ***[indice2***]…[**indiceN**];

Si quisiéramos definir un arreglo para el caso del sobre mencionado en la introducción deberíamos hacerlo de la siguiente manera:

```
int dirección_sobre[edificio][piso][dpto];
```

En el caso de la partícula relativista:

```
float particula[x][y][z][t];
```

Ejemplo: en el código 4.3 se presenta el Cubo de Rubik, imprime las coordenadas de los rectángulos de un mismo color 1-rojo, 2-azul,3-verde, 4-naranja, 5-verde, 6-blanco)

```
#include <stdio.h>
void main()
{
int cubo [6][3][3]={..};  /*Aquí- se deben completar los datos de los cuadros del cubo*/
int i, j,k;                // Se declaran variables índice
for ( i = 0; i < 6 ; i++) //recorre todo el cubo y encuentra los recuadros rojos(1)
{
    for ( j = 0; j < 3 ; j++)
    {
            for (k=0; k<3 ; k++)
            if(cubo[i][j][k]==1)
                    printf("Rectangulo rojo en cara %d, fila %d, columna %d",i,j,k);
    }
}
}
```

Código 4.3. *Cubo de Rubik*

Cadenas de caracteres

Es un tipo particular de arreglos en el cual sus componentes son todos caracteres. Este tipo de arreglos tiene siempre, como último elemento, un carácter especial '\0' (carácter nulo) que determina el final del mismo.

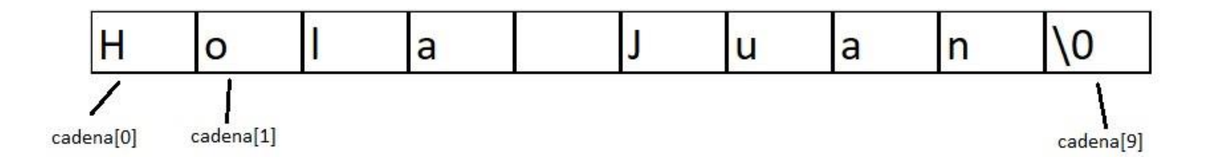
Definición

Tipo_Dato *Nombre_Arreglo* [***Tamaño1+1***]**; // debe reservar un lugar para el carácter nulo**

Inicialización

```
char cadena [10] = “Hola Juan”;
```

Se almacenaría lo siguiente:



El lenguaje de programación C, provee un gran cantidad de funciones específicas para trabajar con cadenas de caracteres que permiten copiar cadenas `strcpy()`, concatenar cadenas `strcat()`, comparar cadenas `strcmp()`, etc. Estas funciones se encuentran agrupadas en la librería string.h.

Existe un tipo especial de arreglos, los arreglos indeterminados que permiten inicializar un arreglo sin determinar su tamaño. El tamaño se asigna automáticamente según la longitud de la cadena asignada.

Ejemplo:

```
char cadena[]=”Este es un arreglo indeterminado”;
```

Si queremos determinar el tamaño total del arreglo `cadena[]`, deberemos utilizar el operador `sizeof()`.

Arreglo de cadenas de caracteres

Son arreglos en los cuales cada fila representa un palabra o cadena de caracteres y la longitud de las mismas lo determina la cantidad de columnas del arreglo.

Definición:

```
Tipo_Dato  Nombre_Arreglo [Tam1][Tam2]={cad1,cad2,cad3….cadTam2-1};
```

Ejemplo:

```
char arreglo_color[3][5]={“azul”,”verde”,”rojo”};
```

Con este tipo de datos se podría conformar un procesador de textos elemental, por ejemplo.

Puntos a remarcar

- Los distintos tipo de datos disponibles permiten la representación de números enteros o caracteres, números reales en simple precisión, números reales en doble precisión; conocidos éstos como tipos de datos nativos o tipos de datos básicos.

- Se puede hacer uso de cada uno de los tipos de datos con una declaración, es decir, especificando su formato a través de las siguientes palabras reservadas: char, int, float, double, void seguidos de un nombre para la variable.

- A partir de los tipos de datos nativos del C, el programador podría construir definiciones más complejas o especializadas con el propósito de la reserva de bloques memoria (ver estructuras – palabra reservada struct).

Algoritmos de búsqueda y ordenación de arreglos

Las acciones de buscar un dato particular entre una colección numerosa de datos, u ordenar un conjunto de datos por algún criterio particular son de las tareas más comunes cuando manipulamos información con una computadora.

Ya sea que la información con que se trabaja esté almacenada en una base de datos, hoja de cálculo, o cualquier otro formato, siempre contamos con funciones que permiten realizar estas búsquedas o reordenamientos de manera automática. En dichas funciones hay implementados algoritmos con métodos conocidos. En la presente sección veremos algunos de los algoritmos más básicos de búsqueda y ordenamiento aplicados a arreglos.

Algoritmos de búsqueda

Los algoritmos de búsqueda tienen como entrada un arreglo (del cual debe conocerse su tamaño) y un valor a buscar en el arreglo (del mismo tipo que el arreglo). La salida de dichos algoritmos es el índice en el que se encuentra el valor buscado. Cuando dicho valor no existe en el arreglo los algoritmos entregan una salida especial para indicar que el valor no fue encontrado. Esta salida especial suele ser un índice inválido como el valor -1.

Búsqueda secuencial

El método de búsqueda secuencial recorre el arreglo elemento por elemento a partir del primero, hasta llegar al final o hasta encontrar el valor buscado. Es de algún modo el mismo procedimiento que llevamos a cabo cuando buscamos un documento particular en una pila de papeles desordenados.

En caso de éxito, la salida del algoritmo será el índice donde se encontró el valor. Mientras que en caso de fracaso se indicará que la búsqueda fue infructuosa entregando un índice inválido (por ejemplo -1).

El código 4.4 muestra una implementación del método de búsqueda secuencial.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  #define MAX 10
5  void main(void)
6  {
7      int elem[MAX]={4,3,11,6,12,6,10,6,11,10}, i, elemento;
8      int posicion; /*valor del indice del elemento*/
9
10     /*ingreso del elemento a buscar*/
11     printf ("Elemento a buscar: ");
12     scanf ("%d",&elemento);
13
14     posicion=-1; /*valor en caso de fracaso*/
15     for( i=0; i<MAX && posicion==-1; i++){
16            if (elemento == elem[i])
17            posicion=i;
18     }
19     if (posicion == -1)
20            printf ("El elemento NO esta en la lista\n");
21     else
22            printf ("Posicion del elemento en la lista: %d\n",posicion);
23 }
```

Código 4.4. *Búsqueda secuencial*

En dicho ejemplo, el usuario ingresa el elemento a buscar (línea 12) en el arreglo de 10 elementos inicializado en tiempo de compilación (línea 7). El bucle (líneas 15-18) recorre elemento por elemento hasta el final o hasta que se encuentra por primera vez el valor buscado y la variable posición deja de valer -1 y pasa a guardar el índice de dicha ocurrencia (línea 17).

La estructura selectiva posterior al bucle verifica si la búsqueda fue exitosa o fracasó, imprimiendo la salida correspondiente en pantalla.

En el mejor de los casos, el elemento buscado está en el primer elemento y solo se realiza una iteración del bucle, mientras que en el peor de los casos, el elemento buscado no se encuentra en el arreglo y el algoritmo debe realizar MAX iteraciones. Por lo tanto, el tiempo de ejecución de este algoritmo, en promedio, crece linealmente con el tamaño del arreglo: O(MAX).

Búsqueda dicotómica o binaria

El método de búsqueda binaria o dicotómica se utiliza para buscar un valor en un arreglo ORDENADO. El método aprovecha que el arreglo está ordenado y en cada iteración verifica si el elemento central contiene el valor buscado, sino divide al arreglo en dos mitades (sub-arreglos):

- una en la que queda descartado que pueda estar el valor buscado
- otra donde puede llegar a estar el valor buscado, y que el algoritmo seguirá procesando en la siguiente iteración,

El algoritmo se ejecuta hasta que el elemento central del arreglo (o sub-arreglo) coincide con el valor buscado o hasta que el último sub-arreglo se queda sin elementos.

Podemos encontrarnos a nosotros mismos aplicando un método similar al de la búsqueda dicotómica cuando buscamos, por ejemplo, los apuntes de una fecha particular en un cuaderno, donde se han tomado apuntes de las distintas clases en orden cronológico. Si la cantidad de clases es grande, en lugar de buscar clase por clase, es siempre más eficiente ir descartando el conjunto de clases entre las cuales, debido a la fecha, es imposible que esté la que buscamos.

El código 4.5 muestra una implementación del método de búsqueda binaria:

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #define MAX 10
4
5  void main(void)
6  {
7      int elem[MAX]= {1,2,6,8,8,10,11,12,12,13}, valor=5;
8      int der, izq, centro; /*indices del arreglo*/
9
10     /*ingreso del elemento a buscar*/
11     printf ("Elemento a buscar: ");
12     scanf ("%d",&valor);
13     der = MAX-1; /*indice más alto*/
14     izq = 0; /*indice más bajo*/
15     centro = ( der + izq ) / 2;
16     while ( izq<=der && elem[centro]!=valor )
17     {
18             if (valor < elem[centro])
19                     der = centro - 1;
20             else
21             izq = centro + 1;
22             centro = (der + izq ) / 2;
23     }
24     if (elem[centro]==valor)
25             printf ("Posición del elemento en la lista: %d\n",centro);
26     else
27             printf ("El elemento NO esta en la lista\n");}/*fin main*/
```

Código 4.5. *Búsqueda binaria*

En dicho ejemplo, el usuario ingresa el elemento a buscar (línea 12) en el arreglo ordenado de 10 elementos inicializado en tiempo de compilación (línea 7). Inicialmente se posicionan los índices `izq` y `der` en el primer y último elemento del arreglo respectivamente (líneas 13-14) y el índice `centro` se calcula como el promedio entero de ambos (línea 15).

El bucle (líneas 16-23) se ejecuta mientras quede un sub-arreglo válido donde seguir buscando `(izq<=der)` y mientras no se haya encontrado el elemento buscado (`elem[centro]!=valor`). Dentro del bucle se parte el sub-arreglo actual en dos, uno donde es válido que esté el valor buscado (será tomado como el sub-arreglo que se evaluará en iteración siguiente) y el otro que se descarta.

Al salir del bucle se debe evaluar el motivo por el cual finalizó este, si fue por encontrar el valor buscado o porque no queda arreglo para evaluar, y se imprime la salida correspondiente en pantalla.

A continuación se muestran dos ejemplos de búsqueda binaria, donde se puede observar la secuencia de partición del arreglo original en sub-arreglos cada vez más pequeños. En el primer caso se busca el 12, que existe en el arreglo. Mientras que en el segundo caso se busca el 8, que no existe en el arreglo.

El elemento correspondiente a los índices `izq` y `der` se marcan en rojo y verde respectivamente. El elemento en el índice centro se subraya.

Valor buscado: 12

iteración:

(1)	3	4	6	6	10	11	12	13
(2)	3	4	6	6	10	11	12	13
(3)	3	4	6	6	10	11	12	13

En este primer ejemplo, en la tercera iteración ya se detecta que el elemento en el índice `centro=6` es igual al valor buscado y esto hace que se interrumpa el bucle de búsqueda.

Valor buscado: 8

iteración:

(1)	3	4	6	6	10	11	12	13
(2)	3	4	6	6	10	11	12	13
(3)	3	4	6	6	10	11	12	13
(4)	3	4	6	6	10	11	12	13

En este ejemplo, en la cuarta iteración, el índice `der=3` se vuelve menor al índice `izq=4`, condición que indica que no queda ningún conjunto de elementos en el arreglo en el cual se pueda encontrar el valor buscado. Esto interrumpe el bucle de búsqueda.

En el mejor de los casos, el elemento buscado está en el centro del arreglo y solo se realiza una iteración del bucle, mientras que en el peor de los casos, el elemento buscado no se encuentra en el arreglo y el algoritmo debe realizar log2(MAX) iteraciones. Por lo tanto, el tiempo de ejecución del algoritmo de búsqueda binaria, en promedio, crece logarítmicamente con el tamaño del arreglo: O(log2(MAX)). Lo cual, para arreglos muy grandes (y ordenados) lo vuelve mucho más rápido que el algoritmo de búsqueda secuencial.

Algoritmos de ordenación *in-situ*

Los algoritmos de ordenación *in-situ* tienen como entrada un arreglo originalmente desordenado (del cual debe conocerse su tamaño) y la salida es el mismo arreglo pero ordenado.

Esto quiere decir que se aprovecha la memoria reservada para el arreglo original, pero se intercambian los valores de los elementos para que, con algún criterio determinado, dichos valores queden ordenados.

Para simplificar la explicación de ahora en más se asumirá que los arreglos a ordenar son numéricos y que el orden buscado es en sentido creciente. Pero esto no quita generalidad a los métodos subsiguientes, que con mínimas modificaciones pueden invertir el criterio de ordenación para que este sea decreciente.

Método de la burbuja o de intercambio directo

El método se basa en ir comparando cada elemento del arreglo con el de la posición siguiente y si están desordenados, se intercambiarán los valores entre ambos elementos.

Si se realiza esto desde el primer elemento del arreglo hasta el anteúltimo, se logra que el mayor valor se ubique al final del mismo, quedando ordenada la parte con los mayores índices del arreglo y desordenados los elementos de menor índice.

Cada vez que se repita lo anterior se incrementa en al menos un elemento a la parte ordenada y se reduce la parte desordenada en la misma cantidad. En MAX iteraciones, como máximo, el arreglo quedará ordenado (siendo MAX el tamaño del arreglo).

La versión más simple de este método se observa en el código 4.6

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #define MAX 8
4  int main()
5  {
6      int arreglo[MAX] = {4,3,11,6,13,1,10,12},aux;
7      int iteracion, i;
8      ...
9      for(iteracion=0; iteracion<MAX; iteracion++){
10             for(i=0; i<MAX-1; i++){
11                     if(arreglo[i]>arreglo[i+1]){
12                         aux = arreglo[i];
13                         arreglo[i] = arreglo[i+1];
14                         arreglo[i+1] = aux;
15             }//fin if
16             }//fin for interno
17     }//fin for externo
18     ...
19     return 0;
20 }
```

Código 4.6. *Método de la burbuja*

En el código se observa el bucle interno (líneas 10-16) que recorre el arreglo desde el primer elemento hasta el anteúltimo y a cada elemento lo compara con el siguiente viendo se están

desordenados (línea 11), de ser esto verdadero intercambia ambos valores haciendo uso de la variable auxiliar `aux` (líneas 12-14). Lo anterior se repite `MAX` veces (el tamaño del arreglo), anidando dicho código dentro del bucle externo (líneas 9-17).

A continuación se ilustra la evolución del contenido del arreglo para las distintas iteraciones del bucle externo. Se identifican en color rojo los elementos que modificaron su valor en la iteración correspondiente y en gris la parte del arreglo ya ordenada:

iteración:								
original	4	3	11	6	13	1	10	12
(0)	3	4	6	11	1	10	12	13
(1)	3	4	6	1	10	11	12	13
(2)	3	4	1	6	10	11	12	13
(3)	3	1	4	6	10	11	12	13
(4)	1	3	4	6	10	11	12	13
(5)	1	3	4	6	10	11	12	13
(6)	1	3	4	6	10	11	12	13
(7)	1	3	4	6	10	11	12	13

Puede verse que hay dos mejoras que se pueden aplicar al algoritmo simplificado presentado anteriormente a fin de reducir el tiempo de ejecución:

- Dado que la parte superior del arreglo va quedando ordenada, y que luego de cada iteración se incrementa en al menos un elemento, se puede reducir el barrido del bucle interno para que recorra el arreglo desde i=0 hasta i=MAX-2-iteración.
- Se puede detectar cuando el arreglo ya está ordenado y finalizar el algoritmo. Para esto se puede utilizar una variable tipo bandera (también llamada centinela) con la cual se detecta si hubo intercambios de valores entre los elementos. De ocurrir toda una iteración del bucle externo sin intercambios implicaría que el arreglo está ordenado, motivo por el cual se detendría el algoritmo.

Aplicando ambas mejoras se obtiene la siguiente versión del algoritmo, donde aparecen resaltadas ambas mejoras:

```
#include <stdio.h>
#include <stdlib.h>
#define MAX 8

int main()
{
    int arreglo[MAX] = {4,3,11,6,13,1,10,12},aux;
    int iteracion, i, intercambios=1;

    ...
    for(iteracion=0; iteracion<MAX && intercambios>0; iteracion++){
            intercambios = 0;
            for(i=0; i<MAX-1-iteracion; i++){
                    if(arreglo[i]>arreglo[i+1]){
                            aux = arreglo[i];
                            arreglo[i] = arreglo[i+1];
                            arreglo[i+1] = aux;
                            intercambios++;
                    }//fin if
            }//fin for interno
    }//fin for externo
    ...
    return 0;
}
```

Código 4.7. *Método de la burbuja mejorado*

Método de selección

El método se basa en buscar el elemento con el menor valor del arreglo e intercambiar con el primer elemento.

Se considera que el primer elemento del arreglo quedó ordenado, entonces se repite el paso anterior con el sub-arreglo que queda a continuación (el cual continúa desordenado).

Cada vez que se repite lo anterior se agrega un nuevo elemento a la parte ordenada (elementos de menor índice del arreglo) y se reduce en un elemento la parte desordenada (mayores índices).

El código 4.8 es un ejemplo que aplica el método de selección para ordenar un arreglo numérico.

```
#include <stdio.h>
#include <stdlib.h>
#define MAX 8

int main()
{
    int arreglo[MAX] = {4,3,11,6,13,1,10,12},aux;
    int i, j, pos_menor;

    ...
    for(i=0; i<MAX-1; i++){
            /* búsqueda del mí-nimo entre
            arreglo[i] y arreglo[MAX-1] */
            pos_menor = i;
            for(j=i+1; j<MAX; j++){
                    if(arreglo[j]<arreglo[pos_menor]){
                    pos_menor=j;
                    }//fin if
```

```
19            }//fin for
20
21            /* intercambio */
22            aux = arreglo[i];
23            arreglo[i]=arreglo[pos_menor];
24            arreglo[pos_menor]=aux;
25     }//fin for
26     ...
27     getchar();
28     return 0;
29 }
```

Código 4.8. Método de selección

A continuación, se ilustra la evolución del contenido del arreglo para las distintas iteraciones del bucle (líneas 11-25). Se identifican en color rojo los elementos que modificaron su valor en la iteración correspondiente y en gris la parte del arreglo ya ordenado:

iteración:								
original	4	3	11	6	13	1	10	12
(0)	1	3	11	6	13	4	10	12
(1)	1	3	11	6	13	4	10	12
(2)	1	3	4	6	13	11	10	12
(3)	1	3	4	6	13	11	10	12
(4)	1	3	4	6	10	11	13	12
(5)	1	3	4	6	10	11	13	12
(6)	1	3	4	6	10	11	12	13

Método de inserción

El método se basa en mantener un subconjunto ordenado en la parte baja del arreglo (índices bajos) y otro subconjunto desordenado en la parte alta (índices altos).

Se recorre el arreglo desde el segundo elemento al último, y para cada elemento evaluado se lo inserta en el lugar del subconjunto ordenado que corresponde

Es un algoritmo conocido para la mayoría de las personas, ya que es generalmente utilizado al ordenar naipes que son tomados de a uno por vez.

A continuación, en el código 4.9 se presenta un ejemplo que ordena el arreglo con el método de inserción.

```
#include <stdio.h>
#include <stdlib.h>
#define MAX 8

int main()
{
```

```
7       int arreglo[MAX] = {4,3,11,6,13,1,10,12},aux;
8       int i, j;
9
10      ...
11      for ( i=1; i < MAX; i++){
12              aux = arreglo[i];
13              j = i -1;
14              while ( aux < arreglo[j] && j >= 0){
15              arreglo[j+1]=arreglo[j];
16              j--;
17              }
18              arreglo[j+1]=aux;
19      }
20      ...
21      getchar();
22      return 0;
23  }
```

Código 4.9. *Método de inserción*

A continuación, se ilustra la evolución del contenido del arreglo para las distintas iteraciones del bucle (líneas 11-19). Se identifican en color rojo los elementos que modificaron su valor en la iteración correspondiente y en gris la parte del arreglo parcialmente ordenada:



iteración:

original	4	3	11	6	13	1	10	12
(1)	3	4	11	6	13	1	10	12
(2)	3	4	11	6	13	1	10	12
(3)	3	4	6	11	13	1	10	12
(4)	3	4	6	11	13	1	10	12
(5)	1	3	4	6	11	13	10	12
(6)	1	3	4	6	10	11	13	12
(7)	1	3	4	6	10	11	12	13

Ejercicios

Arreglos

1) Hacer un programa que:

   a) Declare un arreglo unidimensional de 30 elementos de tipo numérico entero y nombre num.

b) Inicialice el vector en la declaración con valores positivos, negativos y ceros utilizando la notación de llaves { … }.
c) Recorra el arreglo de izquierda a derecha mostrando el contenido de cada elemento, usando alguna estructura de control repetitiva (while, do-while, for).
d) Contabilice en sendas variables el número de valores positivos, negativos y ceros almacenados.
e) Muestre en pantalla los resultados obtenidos.

2) Repita el ejercicio anterior, pero esta vez, en el inciso b, que sea el usuario quien introduzca los valores por teclado.

3) Hacer un programa que:
a) Lea una secuencia de 20 valores numéricos reales y los almacene en un arreglo de nombre “numeros”.
b) Muestre en pantalla cuál es el valor máximo, así como la posición que ocupa en el arreglo. En caso de aparecer repetido el valor máximo se muestra el de menor índice.

4) Cada año un profesor repite el mismo curso, al cual pueden inscribirse hasta un máximo de 60 alumnos. El profesor desea usar un programa donde registrar las notas de cada módulo del curso actual y obtener un promedio. Quede bien con el profesor y escriba el código que permita implementar este programa. Para hacerlo, implemente un arreglo de enteros donde almacenar el número de alumno y una matriz de números reales donde guardar, en cada fila, las notas de cada módulo y en la última columna la nota final.

Arreglo de números de alumno

67852
67923
68123
…

Matriz de notas

5	6.5	…
8.5	9	…
4.5	8	…
…	…	…

Al final, imprima una tabla con el número de alumno, notas parciales y nota final.

Cadenas de caracteres

5) Cree un programa que permita al usuario ingresar una cadena de caracteres y la almacene en el arreglo correspondiente, y luego analice la cadena e imprima en pantalla el análisis que consistirá en encontrar: la cantidad de caracteres (largo de la cadena), cantidad de espacios, de vocales, de consonantes, y de caracteres numéricos.

Nota: Pruebe realizar la lectura de la cadena utilizando las funciones gets(), scanf() y fscanf(). Puede explicar qué sucede al ingresar cadenas más largas que el arreglo donde las almacenará.

6) El siguiente código imprime tres líneas en pantalla
   a) Antes de ejecutarlo, analice el código y prediga la salida que se obtendrá del mismo en pantalla.
   b) Verifique su análisis ejecutando el código.
   c) Interprete los resultados y explique las diferencias.

```
#include <stdio.h>
#define LARGO 11
int main()
{
    char arr[LARGO] = {104,111,108,97,0,109,117,110,100,111,0};
    int i;
    for(i=0;i<LARGO;i++){
        printf("%d ",arr[i]);
    }
    printf("\n");
    for(i=0;i<LARGO;i++){
        printf("%c",arr[i]);
    }
    printf("\n");
    printf("%s",arr);
    printf("\n");
    return 0;
}
```

7) ¿Qué debería modificar en el arreglo del ejercicio anterior para que la expresión printf("%s",arr); imprima “hola mundo”?

8) Implemente el código para convertir un número entero en una cadena con los caracteres que lo representan, de la manera que se muestra en el siguiente ejemplo:

34256 →

‘3’	‘4’	‘2’	‘5’	‘6’	‘\0’		

Realice luego el código para la conversión inversa.

Ordenación y búsqueda

9) Realice un programa que:
   a) Cargue un arreglo con 9 valores enteros.
   b) Los ordene según el método de la burbuja, mostrando en pantalla como se va ordenando el arreglo en cada iteración externa.

10) Cree un programa que inicialice un arreglo numérico, pida un valor al usuario y busque el número en el arreglo utilizando el método dicotómico o binario. Para poder usar el método, primero deberá detectar si el arreglo está ordenado. Si el arreglo no está ordenado, imprima un mensaje de error. Si lo está, pero el valor buscado no está dentro del rango que maneja el arreglo, imprima otro mensaje de error.

CAPÍTULO 5
Funciones

Marcelo A. Haberman

Tal como se introdujo en el capítulo 2, las funciones son el elemento constitutivo de la programación modular. Las mismas permiten la planificación de procesos de desarrollo más eficientes, promueven el trabajo conjunto de varios programadores, la reutilización de código y facilitan el mantenimiento y la depuración.

El lenguaje C no es ajeno al uso de funciones y en el presente capítulo presentaremos la sintaxis apropiada para trabajar con funciones, así como una serie de conceptos nuevos como por ejemplo la declaración de funciones, el ámbito de las variables y la recursividad.

Definición de funciones en C

La sintaxis genérica para la definición de funciones en C, puede verse en el código 5.1.

```
0	<tipo retorno> <NombreFuncion>(<lista de parámetros>)
1	{
2		<código ejecutable>
3	}
```

Código 5.1: *sintaxis genérica para la definición de una función*

La definición se basa en una primera línea (línea 0 en el código 5.1) denominada *cabecera* o *prototipo* de la función y en el bloque de código ejecutable encerrado entre llaves `{ }`, también llamado *cuerpo* de la función.

El **prototipo** de la función cuenta con:

- el **tipo de dato de retorno** que retornará la función. Si la función no retorna nada se utiliza la palabra clave `void`
- el **nombre o identificador** de la función, que sirve para poder invocarla desde otros puntos del programa. Debe seguir las mismas reglas vistas para nombrar variables.
- la **lista de parámetros o argumentos** que recibirá la función, entre paréntesis a continuación del nombre de la función. Cada parámetro es una variable que recibe algún valor de entrada, se define por su tipo y un nombre con el cual se lo puede identificar en el

cuerpo. Los parámetros van separados entre sí por comas. Si la función no recibiese ningún parámetro los paréntesis se dejan vacíos.

El **cuerpo** de la función (líneas 1 a 3 en el código 5.1) consiste en las llaves y las líneas de código encerradas por estas. Este será el código que se ejecute cada vez que se invoque a la función. Si la función devuelve algún valor (el tipo de dato no es `void`) dentro del cuerpo deberá utilizarse la palabra clave `return` acompañada de una expresión del tipo indicado.

Por ejemplo, la función del código 5.2 se identifica por el nombre `Saludar`, no recibe parámetros dado que los paréntesis están vacíos ni tampoco retorna un valor dado que el tipo de la función es `void`.

```
void Saludar()
{
        printf("Función sin parámetros y sin valor de retorno");
}
```

Código 5.2: *Función sin parámetros de entrada ni valor de retorno*

En el código 5.3 se implementa en C la función `Raiz_cuadrada`, definida previamente en pseudocódigo en el código 2.3. En este caso la función devuelve un dato real (es de tipo `float`) y recibe como entrada un parámetro de tipo `float` llamado `x`.

```
0       float Raiz_cuadrada(float x)
1       {
2           float r;
3           const float er = 1e-6;
4
5           r = x;
6           while(fabs(r*r - x) > er*x)
7           {
8                   r = (x/r + r) / 2.0;
9           }
10          return r;
11      }
```

Código 5.3: *Función con un parámetro real y valor de retorno real. La condición del `while` fue modificada para evitar comparar la igualdad de dos expresiones `float`.*

Al igual que en la escritura del programa principal (función `main`), al principio del cuerpo de la función deberán declararse las variables que serán utilizadas luego. En este caso solo se declara la variable de salida `r` en la línea 2 y la constante `er` que representa una franja de error relativo para la condición del bucle. Los parámetros de la función ya se consideran declarados entre los paréntesis y pueden utilizarse directamente. Los mismos contendrán el valor pasado como parámetro a la función en la llamada.

A diferencia de la función del código 5.2, al final del cuerpo de la función del código 5.3 se observa la sentencia `return r;`, la cual implica que el dato de salida de la función es el valor

almacenado en la variable `r`. Para que el proceso de compilación no devuelva errores la expresión que acompaña a `return` debe ser del mismo tipo que la función. En este caso tanto `r` como la función `Raiz_cuadrada` son de tipo float.

En el código 5.4 se observa un ejemplo de una función con más de un parámetro de entrada. En este caso la función calcula y retorna el promedio entre dos números enteros que recibe como parámetro. Nótese que el cálculo del promedio se realiza directamente en la expresión que acompaña al `return`.

```
double Promedio(int a, int b)
{
        return (a + b)*0.5;
}
```

Código 5.4: *Función que calcula y retorna el promedio (real) entre dos números enteros que recibe como parámetro.*

Llamada a funciones en C

Cuando se realiza la llamada a una función, se transfiere el control de la ejecución a dicha función, hasta que la misma finalice o devuelva un resultado mediante la instrucción `return`, volviendo el control de la ejecución a la línea de código de la función que la invocó.

En lenguaje C, la invocación se realiza de manera similar a como lo hicimos en pseudocódigo en el capítulo 2, utilizando el nombre de la función acompañado de un par de paréntesis con los valores pasados como parámetros. Esto mismo es lo que hemos estado haciendo hasta el momento para llamar funciones predefinidas de las bibliotecas de C como `scanf` o `printf`.

Por ejemplo, para invocar a la función `Saludar` del código 5.2 desde `main` o cualquier otra función habría que escribir la siguiente línea:

```
Saludar();
```

En el caso de que la función reciba parámetros de entrada, como `Raiz_cuadrada` o `Promedio` (códigos 5.3 y 5.4), la llamada deberá incluir tantos parámetros como hay en la definición de la función, que para las funciones mencionadas es uno y dos parámetros respectivamente:

```
Raiz_cuadrada(5.01)     /* se pasa un valor real */
Raiz_cuadrada(x+y)      /* se pasa el resultado de una expresión real */
Promedio(10,x)    /* se un valor entero y el valor de una variable entera*/
Promedio(modulo1,modulo2) /* se pasa el valor de dos variables enteras*/
```

Las llamadas anteriores transferirán el control de ejecución a las respectivas funciones, que en cada caso retornarán un valor de salida correspondiente a los parámetros recibidos. El valor de retorno reemplazará a la llamada en cada caso.

Para darle utilidad al valor de retorno, la llamada suele formar parte de una expresión más compleja, por ejemplo:

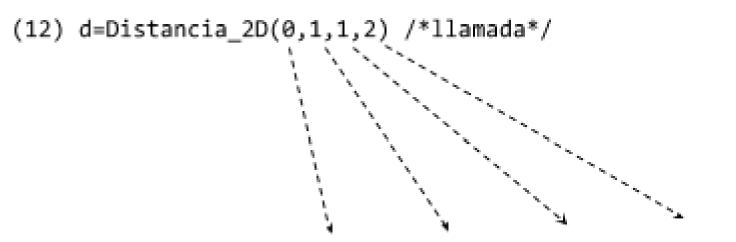
```
/* se almacena el resultado en una variable */
y = Raiz_cuadrada(5.01);
/* se muestra en pantalla el resultado */
printf("%f",Raiz_cuadrada(x+y));
/*el resultado forma parte de una expresión lógica evaluada en un if*/
if(Promedio(modulo1,modulo2) >= 4)
   printf("Felicitaciones, has promocionado!");
```

En el código 5.5 se ve un ejemplo de un programa completo que consta de dos funciones definidas por el programador, la del programa principal (`main`) y la función `Distancia_2D`, que calcula la distancia entre dos puntos en el plano. A su vez, se hace uso de dos funciones predefinidas en las bibliotecas del lenguaje C: `sqrt`, para calcular la raíz cuadrada, y la ya conocida `printf`. Las distintas llamadas han sido resaltadas en el código.

```
0       #include <stdio.h>
1       #include <math.h> /* para poder usar sqrt() */
2
3       float Distancia_2D(float x1, float y1, float x2, float y2)
4       {
5               return sqrt((x1-x2)*(x1-x2) + (y1-y2)*(y1-y2));
6       }
7
8       int main()
9       {
10              float d;
11
12              d = Distancia_2D(0,1,1,2);
13              printf("La distacia entre puntos es: %f", d);
14              return 0;
15      }
```

Código 5.5: *Programa dividido en 2 funciones. Las distintas llamadas a función están resaltadas*

- El programa comienza a ejecutarse por la función `main` y en la línea (12) invoca a la función `Distancia_2D`, pasándole las coordenadas en el plano de los puntos <0,1> y <1,2>.
- En cada llamada a función se reserva el lugar en memoria para almacenar las variables declaradas en la función y los parámetros formales definidos en el prototipo. En el caso de la línea (12) se crean las variables `x1`, `y1`, `x2` e `y2`. Los cuatro valores pasados como parámetros en la llamada se copian o asignan respectivamente a cada una de las cuatro variables definidas como parámetros.




```
(5) Distancia_2D(float x1,float y1,float x2,float y2) /*Definición*/
```

quedando los parámetros formales inicializados de la siguiente manera:

```
x1 = 0     y1 = 1     x2 = 1     y2 = 2
```

- El flujo de ejecución salta a la línea (5), que es la única línea ejecutable de `Distancia_2D`. En esta línea primero se evalúa la expresión del argumento `((x1-x2)*(x1-x2) + (y1-y2)*(y1-y2))`, que para los parámetros pasados a la función vale 2. Luego se invoca a la función `sqrt` pasándole el valor 2 como parámetro.

```
return sqrt((x1-x2)*(x1-x2) + (y1-y2)*(y1-y2));

return sqrt(2);
```

- El control ha sido transferido a la función `sqrt`, que ejecutará algún algoritmo para calcular la raíz cuadrada del parámetro pasado entre paréntesis, en este ejemplo el valor 2. Esta función recibe un valor tipo `double` y retorna también un `double`. En general no contamos con el código fuente, sino que la misma suele venir compilada (en código de máquina) en las bibliotecas que acompañan al compilador.
- Al retornar `sqrt`, la llamada en la línea (5) se reemplaza por el valor retornado (1.41421 en este caso), con lo cual la función `Distancia_2D` retornará el mismo valor que obtuvo de `sqrt`.

```
return sqrt(2);

return 1.41421;
```

- Al retornar `Distancia_2D`, se libera la memoria utilizada para almacenar sus parámetros y variables, y el control del programa vuelve a la línea (12), donde la invocación a la función se reemplaza por el valor retornado (1.41421) y este se asigna directamente a la variable `d`.

```
d = Distancia_2D(0,1,1,2);

d = 1.41421;
```

- En la línea (13) se llama a la función `printf`. Esta escribe en pantalla una serie de caracteres de acuerdo con los parámetros que se pasaron a la función y luego retorna el control a la función que la invocó. En este caso a `main` que finalizará su ejecución con la instrucción de la línea (14), que retorna 0 al sistema operativo.

Diferencia entre parámetros reales y formales

Hay dos instancias en las que mencionamos la utilización de parámetros de una función, en el contexto de la llamada (función invocante) y en la definición de la función invocada.

A los valores pasados como parámetros en el contexto de una llamada a función se los conoce como **parámetros reales** y, para una misma función pueden variar entre distintas invocaciones. Por otro lado, a las variables enumeradas en la lista de parámetros de la definición de una función se las conoce como **parámetros formales**.

Visto de otra manera los **parámetros formales** son las variables donde se copiarán los valores de los **parámetros reales**.

En la invocación de la línea (12) del código 5.5, por ejemplo, se tiene que los parámetros reales son los valores 0, 1, 1 y 2, mientras que los parámetros formales son las variables `x1`, `y1`, `x2` e `y2` donde se almacenarán respectivamente dichos valores.

Pasaje de parámetros por valor

La forma vista de pasar parámetros reales a una función en lenguaje C se denomina "pasaje de parámetros **por valor**", dado que cada parámetro real en una llamada a función es una expresión cuyo valor se copiará al parámetro formal.

Que un parámetro sea pasado por valor tiene un efecto real en la práctica: si el parámetro real de una llamada a función es una variable, esta variable no se verá afectada, aunque se modifique el parámetro formal dentro de la función.

En el código 5.6 puede verse un ejemplo de esto último. En la línea (14) se asigna el valor 10 a la variable entera `v1` y en la línea (14) dicha variable es pasada como parámetro en el llamado a la función `Incremento` y el valor de retorno de esta llamada será almacenado en `v2`.

Dentro de la función `Incremento`, el parámetro formal `x` recibe el valor de `v1`, es decir 10. En la línea (6) la variable `x` es incrementada en uno, por lo cual pasa a valer `11`. Pero, dado que `x` y `v1` son dos variables distintas, `v1` conservará el valor de 10 que tenía previo a la llamada. Finalmente, la función retornará el valor actual de `x`, es decir un 11, el cual será asignado a la variable `v2` y la llamada a `printf` de la linea (14) mostrará en pantalla la leyenda "10 - 11".

```
#include <stdio.h>

int Incremento(int x)    /* x será una nueva variable que guarda
                            una copia del valor entero pasado
                            como parámetro real */
{
    x = x + 1;
      return x;
}

int main()
{
      int v1, v2;

    v1 = 10;             /* Se asigna 10 a v1 */
      v2 = Incremento(v1); /* Se pasa el valor de v1 (que es un
                            entero) como parámetro real a la
                            función Incremento.
                            Se guarda el retorno en v2 */
      printf("%d - %d", v1, v2); /*Imprime: 10 - 11*/
      return 0;
}
```

Código 5.6: *Se muestra un ejemplo donde la función* `Incremento` *tiene un único parámetro por valor.*
Por lo tanto, la función no altera el parámetro real.

Pasaje de parámetros por referencia

El caso opuesto a pasar un parámetro por valor es pasar un parámetro “**por referencia**” que, explicado de forma simple, en lugar de copiar el valor del parámetro real al parámetro formal, copia una “referencia a la variable” pasada como parámetro real. Esto permite modificar el parámetro real desde dentro de la función.

Muchos lenguajes de programación permiten el pasaje de parámetros tanto por valor como por referencia. En C, como venimos viendo, el pasaje de variables simples como parámetros es típicamente por valor, aunque, como se verá en el Capítulo 6, es posible pasar variables por referencia utilizando punteros.

Pasaje de arreglos como parámetros

Cuando un parámetro se indica como arreglo en C, este es tratado como un parámetro por referencia. Esto quiere decir que cualquier modificación realizada a los elementos de un arreglo que sea parámetro formal de una función modificará al elemento correspondiente del parámetro real.

En el código 5.7 se ve un ejemplo de la sintaxis para permitir el pasaje de arreglos como parámetros a funciones. Los parámetros de tipo arreglo se indican en el prototipo de la función

agregando un par de corchetes vacíos `[]` a la derecha del parámetro formal, como sucede en la línea (2). En cuanto al pasaje de un arreglo como parámetro real, este se realiza pasando el nombre del arreglo (sin corchetes), como se muestra en la línea (15).

El lector puede darse una idea de que cuando se trabaja con parámetros por referencia se corre el riesgo de alterar la información original del parámetro real.

```
0       #include <stdio.h>
1
2       int Incremento(int x[]) /* Los corchetes vacíos indican que x
3                                  será una referencia a un arreglo de
4                                  enteros pasado como parámetro real */
5       {
6           x[0] = x[0] + 1; /*Se incrementa el primer elemento de x*/
7           return x[0];     /* Se retorna el elemento incrementado */
8       }
9
10      int main()
11      {
12          int v[2];
13
14          v[0] = 10;           /* Se asigna 10 a v[0] */
15          v[1] = Incremento(v);/* Se pasa v (que es un arreglo de
16                                  enteros) como parámetro real a la
17                                  función Incremento.
18                                  Se guarda el retorno en v[1] */
19          printf("%d - %d", v[0], v[1]); /*Imprime: 11 - 11*/
20          return 0;
21      }
```

Código 5.7: *Se muestra un ejemplo donde la función* `Incremento` *recibe un parámetro por referencia (un arreglo).* Por lo tanto, la función altera el valor del parámetro real.

Organización de los archivos de código

Para que el compilador permita invocar una función desde cualquier otra parte del código, la función debe ser “reconocida” por el compilador, que procesa los archivos de código de arriba hacia abajo. Por este motivo los ejemplos expuestos en los códigos 5.5 a 5.7 presentan la definición de las funciones previo a la definición de `main()` que es dónde se invocan. Esta organización del código se observa en la figura 5.1.

```
archivo .c

#include <...>      } inclusión de archivos de
#define ...         } cabecera y definición de
                    } constantes simbólicas.

void funa(){
  ...
}
                      definición de funciones
void funb(){
  ...
}

main()                definición del programa
{                     principal
  ....
}
```

Figura 5.1: *organización de un archivo de código cuando no se declaran las funciones, sus definiciones están antes de `main`.*

Pero, esto no es lo más práctico porque al abrir un archivo de código debería encontrarse rápidamente el algoritmo principal, codificado en `main()`, además de obligar al programador a tener en cuenta la posible jerarquía de llamadas entre las distintas funciones para ordenar sus definiciones en el orden correcto.

Declaración de funciones

Para organizar el código de una manera más ventajosa, existe el concepto de *declaración de una función* en el cual, le decimos al compilador que existe una función con un identificador determinado, le informamos el tipo de valor de retorno y la cantidad de parámetros y sus tipos. Esto lo hacemos con una línea de código con el prototipo de la función, terminada en punto y coma.

```
<tipo retorno> <NombreFuncion>(<lista de parámetros>);
```

Lo que no se incluye en esta instancia de declaración es el cuerpo de la función, ya que este se incluirá con la definición.

Un archivo de código mejor organizado incluirá las declaraciones de las funciones antes del programa principal y todas las definiciones después, como se puede observar en la figura 5.2.

archivo .c

```
#include <...>
#define ...
```
inclusión de archivos de cabecera y definición de constantes simbólicas

void funa();
void funb();
declaración de funciones

```
main()
{
  ....
}
```
definición del programa principal

```
void funa(){
  ...
}

void funb(){
  ...
}
```
definición de funciones

Figura 5.2: *organización de un archivo de código cuando se declaran las funciones,* puede definirse `main` antes que el resto haciendo más legible el código.

Por ejemplo, el código 5.7 puede reorganizarse incluyendo la declaración de la función `Incremento` como se ve en el código 5.8.

```
#include <stdio.h>
int Incremento(int x[]); /* Declaración de la función Incremento
                     que recibe un arreglo de enteros y devuelve un entero*/
int main() /*Definición de la función main*/
{
    int v[2];

    v[0] = 10;           /* Se asigna 10 a v[0] */
    v[1] = Incremento(v);/* Se pasa v (que es un arreglo de
                            enteros) como parámetro real a la
                            función Incremento.
                            Se guarda el retorno en v[1] */
    printf("%d - %d", v[0], v[1]); /*Imprime: 11 - 11*/
    return 0;
}
int Incremento(int x[])/*Definición de la función Incremento*/

{
    x[0] = x[0] + 1; /*Se incrementa el primer elemento de x*/
    return x[0];     /* Se retorna el elemento incrementado */
}
```

Código 5.8: *el mismo código del código 5.7 pero organizado de manera que `main` sea la primera función definida. Para esto es necesario declarar la función `Incremento` al principio.*

En la declaración no es necesario dar el nombre de los parámetros formales, solo su tipo y el orden correcto, por lo que la definición de `Incremento()` en el código 5.8 podría escribirse también como: `int Incremento(int []);`

La inclusión de los archivos de cabeceras (.h) mediante la directiva `#include` lo que hace es insertar, en el lugar de la directiva, el contenido de dichos archivos, los cuales suelen ser un largo listádo de declaraciones de funciones. Por ejemplo el archivo `stdio.h` tiene, entre otras, las declaraciones de `printf()` y `scanf()`. Al incluir `stdio.h`, el compilador nos deja invocar dichas funciones, chequeando que la cantidad de parámetros y los tipos sean apropiados. El código objeto de dichas funciones (que ya está previamente compilado por ser funciones de la librería estándar de C) recién se agregará al código objeto del programa principal en la etapa de enlazado, creando un archivo ejecutable (.exe).

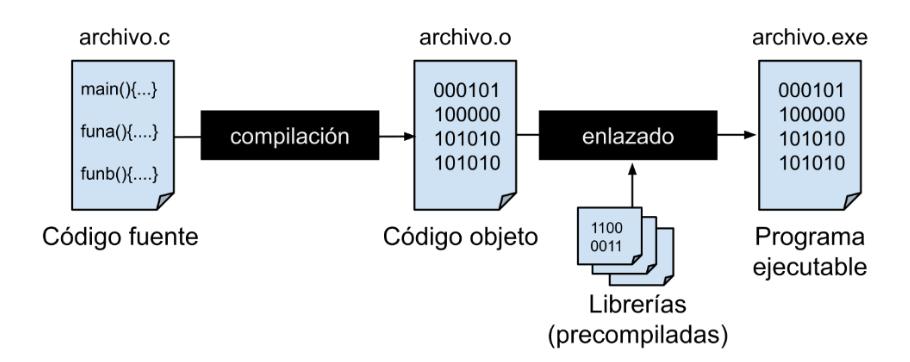


Figura 5.3: *Proceso de construcción de un programa ejecutable a partir de un único archivo fuente.*

Proyectos con múltiples archivos de código

Hasta el momento hemos venido compilando proyectos que incluyen un único archivo fuente. En el escenario más amplio implementado hasta ahora hemos agregado varios módulos (funciones) a este archivo, pero siempre ha sido en un único archivo de código.

Cuando los proyectos son grandes y se necesitan implementar muchos módulos, la buena práctica de programación indica que conviene separar los módulos en distintos archivos fuente, agrupándolos por su función. Así, además del archivo fuente que incluye el programa principal, podrán existir un archivo (módulo 1) que incluya las funciones matemáticas, otro que incluya las funciones para la interfaz de usuario (módulo 2), misceláneos (módulo 3), etc.

El proceso de compilación se realiza independientemente en cada archivo fuente, generando un archivo en código objeto (.o) por cada archivo fuente (.c). Si el proyecto se trata de un programa ejecutable, en alguno de los archivos fuente deberá existir la definición de `main`; en el caso de que el proyecto sea una librería precompilada no es necesario. Desde un archivo fuente podrán invocarse funciones definidas en otros archivos, siempre y cuando la función esté previamente declarada en el archivo que realiza la invocación.

En la etapa de enlazado se junta el código objeto de todas las funciones definidas en los distintos módulos, se resuelven las llamadas entre funciones de distintos módulos y las de funciones de la librería estándar (u otras librerías utilizadas).

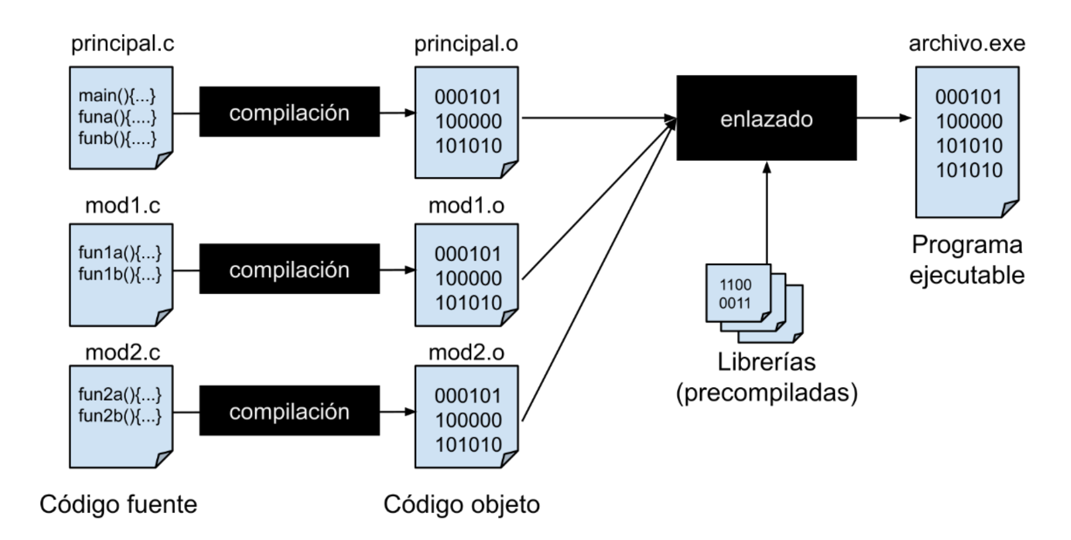


Figura 5.4: *Proceso de construcción (compilación+enlazado) de un programa ejecutable a partir de varios archivos fuente.*

Ámbito de las variables

Se denomina ámbito de una variable a las partes de un programa donde una variable es conocida y accesible. Dependiendo del lugar en el código dónde se defina una variable, esta será visible o no por los distintos módulos. La distinción más tajante la haremos entre **variables globales** y **variables locales**.

Variables locales

Esta categoría corresponde a las variables que han sido declaradas en el interior de una función, o a los parámetros formales de una función. El ámbito de una variable local es la función o bloque de código donde se declara, no pudiendo ser manipulada por ninguna otra función de manera directa (es posible a través de punteros, tema que se tratará en el capítulo 6). Dentro de las variables locales debemos hacer una distinción entre las variables locales **automáticas** y las **estáticas**.

Las **automáticas** son el tipo de variable local que hemos estado utilizando hasta el momento, incluyendo los parámetros formales. Estas se crean en una zona de memoria denominada pila o *stack* al ser invocada la función y se destruyen al finalizar la ejecución misma (por lo que no conservan el valor de una llamada a la siguiente).

Las variables locales **estáticas** son variables que se declaran anteponiendo el modificador `static` y, a diferencia de las automáticas, conservan el valor entre distintas invocaciones de la función. Estas se crean en una zona de memoria distinta del *stack*, que perdura durante toda la ejecución del programa.

Variables globales

Son las variables declaradas por fuera de cualquier función, en general, luego de la sección de declaración de funciones. Estas variables pueden ser accedidas desde cualquier función cuya definición ocurra después de la declaración de la variable.

Cuando una variable local tiene el mismo nombre que una global, tiene prioridad la local, dentro de su ámbito y la global se vuelve inaccesible ya que queda “enmascarada” por la local.

En el código 5.9 puede verse que existen dos variables globales, `c` declarada en la línea 3 y `d` en la línea 14. Lo que diferencia a ambas en principio es que `c` será visible por todas las funciones del archivo fuente, pero `d` solo por la función `f()`, ya que está declarada a continuación de `main()`.

Sin embargo, en la línea 6 encontramos la declaración de las variables `a`, `b`, `c` y `d`, locales a `main()`, por lo que la variable global `c` también se hace inaccesible desde `main()`. Estas cuatro variables locales a `main()` se inicializan en cero en la línea 7, y en las líneas 8 y 12 se imprimen en pantalla. Como no se ven modificadas se imprimirán con sus valores en cero. Mientras tanto, la función f() tiene dos variables locales, `a` y `b` que se inicializan con 10, siendo la primera estática y la segunda automática. En la línea 18 la función imprime el estado de las dos variables locales y las dos globales. Luego, entre la línea 19 y 22 se incrementan en 1 las cuatro variables.

La función principal invoca tres veces a `f()`. En la línea 9, al ejecutarse f() por primera vez se inicializan `a` y `b` con 10 y se imprimen ambas variables globales (todas en 10), luego se incrementan en 1 y la función termina. Al finalizar la función la variable automática `b` se destruye, mientras que `a`, que es estática, persistirá en memoria (al igual que las globales `c` y `d`). Al ser invocada `f()` nuevamente en las líneas 10 y 11, la inicialización de `a` en la línea 16 es ignorada, mientras que la de `b` en la línea 17 se repite en cada invocación, con lo cual siempre vuelve al valor 10.

```
1   #include <stdio.h>
2   void f();
3   int c=10;
4   int main()
5   {
6       int a,b,c,d;
7       a=b=c=d=0;
8       printf("a=%d b=%d c=%d d=%d\n",a,b,c,d);
9       f();
10      f();
11      f();
12      printf("a=%d b=%d c=%d d=%d\n",a,b,c,d);
13  }
14  int d=10;
15  void f(){
16      static int a=10;
17      int b = 10;
18      printf("a=%d b=%d c=%d d=%d\n",a,b,c,d);
19      a++;
20      b++;
21      c++;
22      d++;
23  }
```

```
a=0 b=0 c=0 d=0
a=10 b=10 c=10 d=10
a=11 b=10 c=11 d=11
a=12 b=10 c=12 d=12
a=0 b=0 c=0 d=0
```

Código 5.9: *Ejemplo que permite visualizar variables en distintos ámbitos.*

Variables globales en proyectos con múltiples archivos fuente

Si se trabaja con múltiples archivos fuente, una variable global definida en un archivo puede ser accedida desde cualquier otro, siempre y cuando en los otros archivos esté re-declarada antes de su uso con el modificador `extern`.

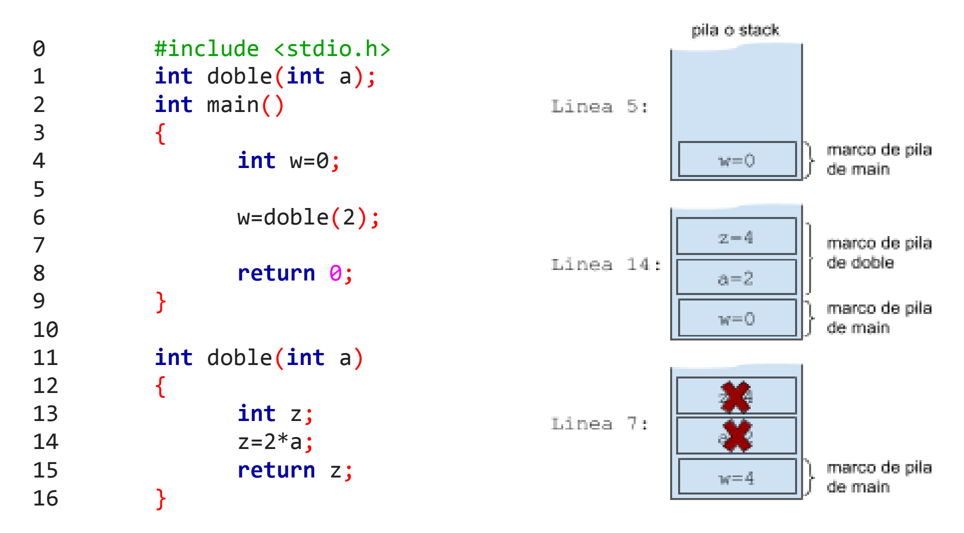
Para que una variable global no pueda ser utilizada desde otro archivo de código esta debe definirse con el modificador `static`. En este caso la variable sólo será visible por las funciones del mismo archivo fuente que estén definidas luego de la variable.

Gestión de la memoria

Cada llamada a una función hace uso de una zona de memoria conocida como *pila de ejecución*, o simplemente *pila* o *stack*.

Cada vez que se invoca a una función, se consume nueva memoria del *stack* para almacenar los parámetros pasados a la función y las variables locales automáticas. A este segmento de memoria correspondiente a una llamada se lo denomina *marco de pila* y en este también se guardan la dirección de memoria de la instrucción ejecutable del programa que deberá ejecutarse al retornar la función y valores intermedios de algunas operaciones aritméticas. Cuando una función retorna, la memoria ocupada por su marco de pila se libera, destruyéndose todas sus variables locales automáticas y parámetros.

Dicho de manera más simple podemos interpretar que al llamar una función se "apilan" en la parte superior del *stack* las variables (locales automáticas y parámetros) de la función, y al retornar una función estas variables se "desapilan". Este mecanismo se observa en la figura que acompaña al código 5.10, dónde hasta la línea 6 solo existe en la pila el marco de la función `main` con la variable `w`, mientras se ejecuta la función `doble` se apilan las variables `a` y `z`, las cuales se desapilan al retornar esta función `y volver a` `main`.



Código 5.10: *Ejemplo donde se observa la evolución de la pila a medida que progresa la ejecución del programa*

Análisis y traza de programas modulares

Si bien la traza es una buena herramienta para analizar código a mano con hoja y papel, la herramienta que se suele utilizar con este fin es el *debugger* o *depurador.*

Realizar la traza de un programa modular consiste en analizar línea por línea el estado de las variables que intervienen en el algoritmo llevando registro de su movimiento. Para esto debemos en principio numerar las líneas de código a analizar y construir una tabla donde incluimos los números de línea, en la secuencia en que las mismas son ejecutadas, y las variables a analizar.

En el caso del código 5.11 se ejemplifica con un programa que utiliza variables globales y tres funciones. El punto de entrada para el análisis será la función `main`, desde donde arranca el programa.

```
1       #include <stdio.h>
2       #include <stdlib.h>
3       int a,b;
4       void funcion1(int c);
5       int funcion2(int d, int e);
6
7       int main()
8       {
9            int g=10;
10           b=3;
11           a=5;
12           funcion1(g);
13           b=funcion2(g,8);
14           printf("a=%d b=%d g=%d",a,b,g);
15           return 0;
16      }
17
18      void funcion1(int c)
19      {
20           int b;
21           b=2;
22           a=a+b+c;
23           return;
24      }
25      int funcion2(int d,int e)
26      {
27           int f;
28           f=a+d+e;
29           return f;
30      }
```

Var	Globales		main	funcion1		funcion2		
línea	a	b	g	c	b	d	e	f
9	?	?	10					
10	?	3	10					
11	5	3	10					
12	5	3	10	10	?			
21	5	3	10	10	2			
22	17	3	10	10	2			
13	17	3	10			10	8	?
28	17	3	10			10	8	35
14	17	35	10					

Código 5.11: *Ejemplo de programa modular con tres funciones y su traza. Se marcan las líneas de código ejecutable que se incluyen en la traza.*

En la traza correspondiente al código 5.11 puede observarse que las variables del programa se agrupan según sean estas globales o locales a una función particular. Cuando las funciones no están en ejecución (no existe un marco de pila en el *stack*), sus variables locales automáticas no existen y se dejan vacías. Si las variables existen, pero no han sido inicializadas se completan

con un signo de interrogación (?). Los valores resaltados permiten visualizar más fácilmente los cambios en las variables.

Recursividad

La recursividad es una técnica que permite definir problemas en términos de sí mismos. Aplicada a la programación se vuelve una técnica muy poderosa, en la cual una función realiza llamadas a sí misma para resolver un problema. A estas funciones se las denomina **funciones recursivas**.

Prácticamente cualquier problema resuelto con estructuras iterativas puede ser resuelto también por funciones recursivas. Los principales motivos para utilizar funciones recursivas son:

- Problemas “casi” irresolubles con las estructuras iterativas.
- Soluciones elegantes.
- Soluciones más simples.

Pero no todas son ventajas, el costo a pagar con el uso de funciones recursivas es un consumo mucho mayor de memoria y mayores tiempos de ejecución.

En las funciones recursivas, cuando están bien definidas, se identifican dos elementos:

- Caso Recursivo: aquí la función realiza algunas operaciones con las que se reduce la complejidad del problema y luego realiza un llamado a sí misma.
- Caso Base: Se da cuando el cálculo es tan simple que se puede resolver directamente sin necesidad de hacer una llamada recursiva.

Un ejemplo típicamente citado para explicar la recursividad es la definición de una función que retorne el factorial de un número recibido como parámetro. Recordemos que el factorial de cualquier número natural está definido como:

$$n! = n \cdot (n-1) \cdot (n-2) \cdots 2 \cdot 1 \qquad (\forall\, n \in \mathbb{N}; n > 0)$$

Pero, observando el segundo término, podemos reescribir dicha definición

$$n! = n \cdot \underbrace{(n-1) \cdot (n-2) \cdots 2 \cdot 1}_{(n-1)!}$$

$$\boldsymbol{n}! = n \cdot (\boldsymbol{n} - \boldsymbol{1})!$$

Esta es la definición recursiva de la operación factorial, que debería valer para cualquier entero positivo. Sin embargo, para terminar con la recursión de cualquier función es necesario establecer el caso base. Para el factorial, el caso base es:

$$\boldsymbol{0}! = \boldsymbol{1}$$

Considerando los mencionados casos recursivo y base, en el código 5.12 se define la función recursiva para el cálculo del factorial.

```
unsigned int factorial (unsigned int n)
{
    if(n==0)
            return 1; /*caso base*/
    else
            return n*factorial(n-1); /*caso recursivo*/
}
```

Código 5.12: *Función recursiva que calcula el factorial de un número natural.*

Análisis de llamadas a funciones recursivas

Una función recursiva se invocará inicialmente desde otra función (como una función normal) y esto desencadenará una serie de llamadas recursivas, que en general dependerá de los parámetros pasados en la llamada original. Por lo tanto, la traza vista para funciones no recursivas no es práctica ya que habría que incluir una nueva columna por cada llamado a la función. En su lugar puede resultar más útil analizar el *stack* considerando que cada llamada recursiva apilará un nuevo conjunto de variables sobre la instancia anterior de la función.

En la Tabla 5.1 se muestra el análisis del desarrollo de las llamadas recursivas a la función `factorial` del código 5.12, a partir de ser invocada desde el siguiente programa principal:

```
int main()
{
    unsigned int resultado;
    ...
    resultado=factorial(4);
    ...
}
```

Tabla 5.1: Análisis de la ejecución de una función recursiva

Instancia en ejecución	Descripción	Estado de la pila
`main()`	Solo existe el marco de pila de `main`. Pero, al momento de realizar la primera llamada, se crea el marco de pila para la llamada `factorial(4)`, que comienza a ejecutarse.	stack resultado=? } main
`factorial(4)`	como `n=4` es distinto de cero, estamos en el **caso recursivo** y la función retornará `4*factorial(3)`, por lo que se crea un nuevo marco de pila para la nueva llamada recursiva y se comienza a ejecutar la nueva instancia de la función, esta vez con `n=3`:	stack n=4 } factorial(4) resultado=? } main

`factorial(3)`	Sigue el **caso recursivo** y la función retornará `3*factorial(2)`, creándose un nuevo marco de pila para la nueva llamada recursiva y se comienza a ejecutar la nueva instancia de la función, esta vez con `n=2`:
`factorial(2)`	Se repite el **caso recursivo**. La función retornará `2*factorial(1)`, creándose un nuevo marco de pila para la nueva llamada recursiva y se comienza a ejecutar la nueva instancia de la función, esta vez con `n=1`
`factorial(1)`	Continua el **caso recursivo** y la función retornará `1*factorial(0)`, con lo cual se crea un nuevo marco de pila para la nueva llamada recursiva y se comienza a ejecutar la nueva instancia de la función, esta vez con `n=0`: …
`factorial(0)`	Se alcanza el **caso base**, por lo que la función retorna el valor 1 y se desapila del *stack*, volviendo el control de la ejecución a la instancia anterior (`factorial(1)`).
`factorial(1)`	La ejecución vuelve a la línea donde se invocó `factorial(0)`, que se reemplaza por el valor retornado, por lo que la función retorna el valor `1*1=1` y se desapila del *stack*, volviendo el control de la ejecución a la instancia anterior (`factorial(2)`).
`factorial(2)`	La ejecución vuelve a la línea donde se invocó `factorial(1)`, que se reemplaza por el valor retornado, por lo que la función retorna el valor `2*1=2` y se desapila del *stack*, volviendo el control de la ejecución a la instancia anterior (`factorial(3)`).
`factorial(3)`	La ejecución vuelve a la línea donde se invocó `factorial(2)`, que se reemplaza por el valor retornado, por lo que la función retorna el valor `3*2=6` y se desapila del *stack*, volviendo el control de la ejecución a la instancia anterior (`factorial(4)`).
`factorial(4)`	Prosiguiendo con la misma lógica, la instancia actual devolverá `4*6=24` a la función principal …
`main()`	El programa principal almacena el valor retornado por `factorial(4)` en la variable `resultado` y continuará su ejecución.

Parámetros de la función main

Nota: Este es un tema avanzado que normalmente se da en el curso luego de ver punteros.

Como se ha tratado anteriormente, la función `main` representa el punto de entrada de todo programa e implementa el algoritmo principal del mismo, el de más alto nivel de abstracción. Esta es una función especial que no suele invocarse directamente desde el código, sino que es invocada por el sistema operativo cuando se requiere la ejecución del programa[13].

Los sistemas operativos ofrecen distintos mecanismos por los cuales un usuario puede iniciar la ejecución de un programa. Uno de estos es hacer "click" o "doble click" con el mouse sobre el ícono de un programa o acceso directo al mismo, mientras que otro mecanismo más clásico es la línea de comandos, consola o terminal del sistema[14], donde el usuario escribe el nombre del programa a ejecutar con el teclado. En Windows, además se cuenta con el diálogo "Ejecutar" que permite escribir el nombre del programa (ver Figura 5.5).

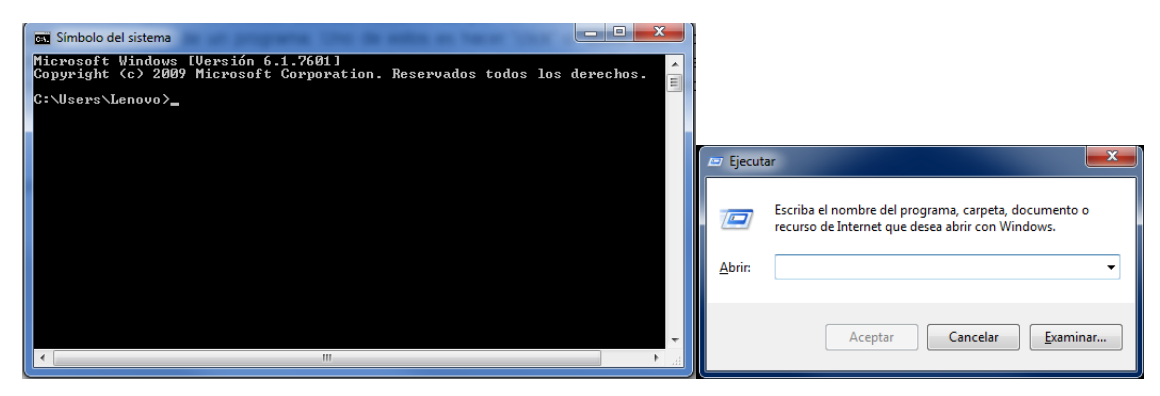


Figura 5.5: *Consola del sistema y diálogo "Ejecutar" de Windows 7.*

Para agilizar la interacción del usuario con el programa, y también para automatizar ciertas tareas, es posible pasarle información de entrada a un programa al momento de solicitar su ejecución. Esta información se da en forma de una o más cadenas de caracteres escritas a continuación del nombre del ejecutable y separadas por espacios o tabulaciones. Si se quisiera pasar una cadena que internamente tiene espacios deberá encerrarse entre comillas dobles (" "). El nombre del ejecutable suele ser considerado como el primer parámetro.

Cada una de estas cadenas puede ser accedida desde dentro de la función `main` gracias dos parámetros especiales con los que cuenta esta función: `argc` y `argv`. Para eso, la función `main` debe ser definida con el siguiente prototipo:

```
int main(int argc, char* argv[])
```

El primer argumento, `argc` es un entero que cuenta la cantidad de parámetros pasados al programa y el segundo, `argv`, es un arreglo con las cadenas de caracteres pasadas por el sistema operativo al programa en el momento de su invocación. Hay que recordar que el nombre

[13] En el caso de no existir un sistema operativo, como es el caso de algunos sistemas embebidos, la función `main` comenzará a ejecutarse al encender o reiniciar el sistema.
[14] En windows se denomina "símbolo del sistema".

del programa ejecutable siempre es considerado como primer parámetro (`argc` será siempre mayor o igual a 1) y se accede mediante la cadena `argv[0]`.

En el siguiente ejemplo se ve un programa que lista en pantalla los parámetros recibidos.

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc,char* argv[])
{
      int i;
     printf("para esta corrida argc vale: %d\n",argc);
     for(i=0;i<argc;i++)
        printf("argv[%d] vale: %s\n",i,argv[i]);
      return 0;
}
```

Código 5.13: *Programa que muestra la lista de parámetros pasados por el SO en la invocación.*

En la figura 5.6 se muestra la salida del programa del código 5.13 que fue compilado y cuyo archivo ejecutable se llama “lista.exe”. Al mismo se lo invoca varias veces con distinta cantidad de parámetros.

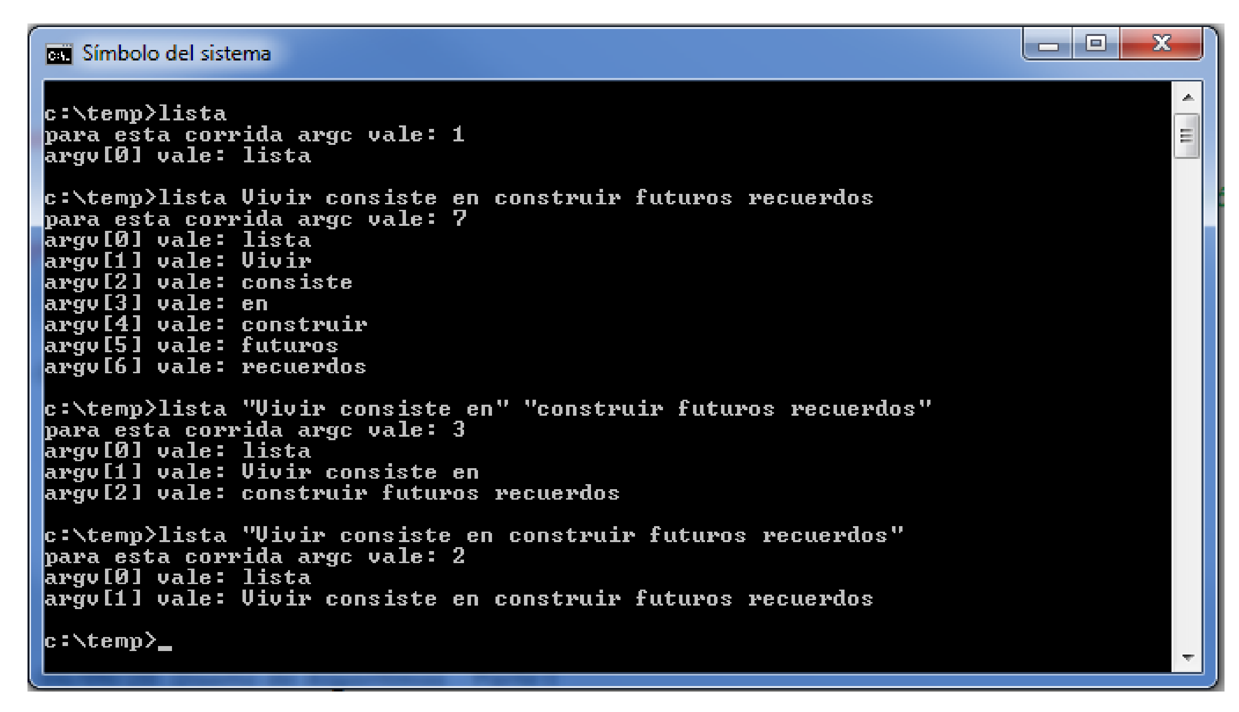


Figura 5. 6: *Distintas salidas del programa del Código 5.13, para distintas invocaciones con diferentes parámetros.*

En el caso de necesitarse el uso de parámetros numéricos pueden usarse las siguientes funciones (declaradas en `stdlib.h`) que convierten las cadenas de caracteres ASCII a `double`, `long` e `int`, respectivamente:

```
double atof(const char *numPtr);
long int atol(const char *numPtr);
int atoi(const char *numPtr);
```

En el siguiente ejemplo se muestra un programa que recibe un conjunto de números reales como parámetro y calcula su promedio, imprimiéndolo en la pantalla.

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc,char* argv[])
{
      int i;
      double suma=0;
      if(argc==1)
      {
          printf("No se ingresaron datos\n");
          exit(1);
      }
      for(i=1;i<argc;i++)
      suma += atof(argv[i]);
      printf("%lf",suma/(argc-1));
      return 0;
}
```

Código 5.14: *Programa que calcula el promedio de los números pasados como parámetros*

En la figura 5.7 se muestran distintas salidas del código 5.14 (compilado con el nombre prom.exe) para distintas ejecuciones con diferentes parámetros.

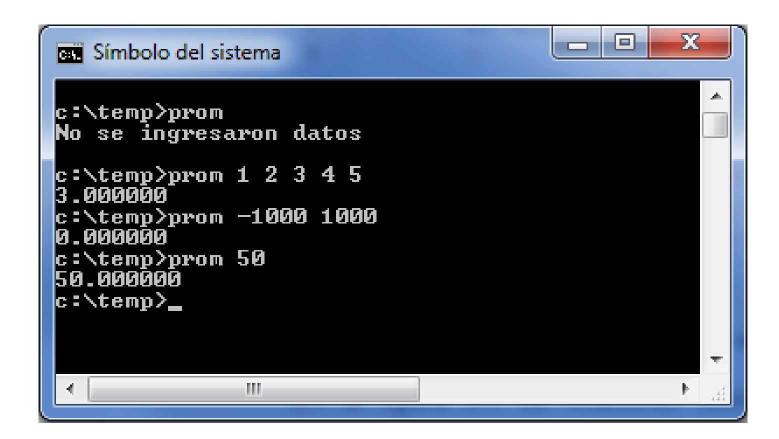


Figura 5. 7: *Distintas salidas del programa del Código 5.14, para distintas invocaciones con diferentes parámetros.*

Ejercicios

Manejo básico de funciones

1) Escriba una función que imprima en pantalla el cartel "Hola mundo desde una función".

Identifique claramente:

a) La declaración de la función
b) La definición de la función
c) El tipo de retorno
d) El tipo de argumento

2) Escriba una función que tome dos números enteros como argumento y devuelva la suma de ambos como valor de retorno.
   a) Repita los incisos del ejercicio 1 para esta función.
   b) Desde el cuerpo de `main()`, pásele dos valores e imprima en pantalla el resultado.

3) Escriba una función que tome dos números enteros y un carácter como argumento. El valor de retorno debe ser un cálculo efectuado sobre los dos números según indique el carácter, que podrá ser '+', '-', '*' o '/'. Usar la estructura de control `switch`.

4) ¿Cuáles de estos conjuntos de declaración, llamada y definición son correctos? En caso de encontrar errores, corríjalos.

(a)		(b)	
Declaración	`void func1(int a, int b);`	Declaración	`double func2(char a, float b);`
Llamada	`func1(x,y);`	Llamada	`double v;` `v=func2(2.3, 'h');`
Definición	`void func1(int p, int q){` `  …` `  return;` `}`	Definición	`double func2(char a, float b){` `  double x;` `  …` `  return x;` `}`

(c)		(d)	
Declaración	`double func3(int, int);`	Declaración	`char func4( void);`
Llamada	`func3(3, 24);`	Llamada	`char c;` `c=func4();`
Definición	`double func3(int y, int z){` `  double x;` `  …` `  return x;` `}`	Definición	`char func4( void){` `…` `return;` `}`

5) Realice el programa que permita imprimir la figura que se muestra más abajo. El usuario ingresará por teclado la cantidad de filas y los caracteres para hacer el dibujo. En los siguientes ejemplos se indica el número de filas y los dos caracteres que el usuario ingresa en cada caso. *Notar que en el último ingresa un espacio ' '.*

4 filas, ‘x‘, ‘o’	3 filas, ‘-’, ‘$’	5 filas, ‘ ‘,’*’
xxxo	--$	*
xxoo	-$$	**
xooo	$$$	***
oooo		****
		*****

El programa principal deberá ser modular y utilizar las siguientes funciones, que también deberá implementar:

a) Una función que imprima un caracter repetidas veces. En particular, el carácter se pasará como primer parámetro y las veces como segundo parámetro:

```
void ImprimirCaracteres(char caracter, int veces);
```

b) Una función que imprima una fila de la figura, lo cual implica imprimir un carácter a una cantidad na de veces y otro carácter b una cantidad nb de veces:

```
void ImprimirFila(char a, int na, char b, char nb);
```

Esta función utilizará a su vez la función ImprimirCaracteres para simplificar su implementación

6) Pase el programa del ejercicio 5 a una nueva función:

```
void ImprimirPiramide(int filas, char caracter);
```

7) Escriba una función de nombre puntoscorte( ), que tenga como parámetros los centros y radios de dos circunferencias y retorne los puntos que tienen en común dichas circunferencias (ninguno, uno, dos o infinitos).

Ámbito de las variables

8) Dados el siguiente programa, indique las salidas por pantalla que genera.

```
int x;
void Calculo();
int main()
{
   x = 10;
   Calculo();
   printf(“x = %d\n”, x);
   return 0;
}
void Calculo()
{
   int x;
   x = x + 7;
   return;
}
```

9) Dado el siguiente programa, seleccione la salida correcta.

```
int x;
int f(int n);
int main( )
{  x = 6;
   printf("%d %d", f(x), x);
}
int f(int n)
{
   n = n + 4;
   return n;
}
```

a) 10 10

Recursividad

10) La función `esimpar()` devuelve 1 si el número que se le pasa es impar y 0 si es par. Observe la implementación recursiva de la función y

a) Encuentre la condición de parada o "caso base"

b) Realice la traza de la llamada `esimpar(7)`

c) Escriba la versión iterativa de la función

```
int esimpar(int n){
    if(n<2)
        return n;
    else
        return esimpar(n-2);
}
```

11) Llame a la función `piramrecu(5)` implementada primero con el código de la izquierda y luego con el de la derecha y explique la diferencia drástica entre las dos

```
void piramrecu(int n){
    int i;
    if(n<=0){
        return;
    }
    else{
        piramrecu(n-1);

for(i=0;i<n;i++){

printf("*");
        }
        printf("\n");

    }
}
```

```
void piramrecu(int n){
    int i;
    if(n<=0){
        return;
    }
    else{

for(i=0;i<n;i++){

printf("*");
        }
        printf("\n");
        piramrecu(n-1);
    }
}
```

12) Escribir una función recursiva que muestre en pantalla los dígitos de un valor numérico en orden inverso. Modifíquela para que los escriba en el orden correcto. *Nota: es recomendable comenzar por la versión "iterativa" de la solución*

.

13) Calcular el término enésimo de la sucesión de Fibonacci en forma recursiva.
Nota: la sucesión de Fibonacci es 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, ...
es decir que $Fibo_n = Fibo_{n-1} + Fibo_{n-2}$

Funciones y Arreglos

14) Realice una función que sea capaz de recibir un arreglo de números reales y retornar el promedio de los valores.

15) Se tienen las siguientes funciones:

```
int duplicar_var(int a){
    return a*2;
}
void duplicar_arr(int a[], int n){
    int i;
    for(i=0;i<n; i++){
        a[i] = 2*a[i];
}
}
```

a) Cree un programa donde demuestre el uso de ambas.
b) ¿Por qué es necesario pasar un segundo argumento en la función duplicar_arr? (a la luz de esto, ¿hizo bien el ej. 14?) ¿Cambiaría esto si supiese que el arreglo se trata de una cadena de caracteres?
c) ¿Por qué la función duplicar_arr puede ser void mientras que la función duplicar_var debe ser int ?

16) Cree una función que pueda detectar si un arreglo de enteros está ordenado. La función recibirá tres argumentos: un arreglo de enteros, su largo, y un caracter. Si el carácter es 'c' la función intentará determinar si el arreglo está ordenado en forma creciente y retornará 1 si lo está y 0 si no lo está. Si el caracter es 'd' , hará lo mismo pero determinando si está ordenado en forma decreciente. *Nota: Para determinar si un arreglo está desordenado basta con detectar si cualquier par de elementos consecutivos están desordenados*

Parámetros de la función main

17) Cree un proyecto y compile el programa "hola mundo", luego ejecútelo desde la línea de comandos de su sistema operativo.

18) Cree un programa capaz de imprimir en pantalla el número de argumentos que se le pasaron al main, y compruebe que funcione ejecutándolo varias veces con distinta cantidad de argumentos desde la línea de comandos. Pruebe luego ejecutarlo desde el CodeBlocks introduciendo los argumentos en el campo que el IDE tiene previsto para simular entradas de argumentos del main().

19) Escriba un programa que pueda dibujar en pantalla distintas figuras (cuadrado, pirámide y una X al menos) según se le pase el argumento "f1", "f2", etc. Además, se debe introducir como argumento el tamaño de la figura. Si el comando no es el adecuado, debe imprimir en pantalla "Comando incorrect. Introduzca 'h' para la ayuda." Si se introduce "h" como argumento, debe imprimir una ayuda con los comandos disponibles.

CAPÍTULO 6
Punteros

Federico N. Guerrero

Introducción

Un puntero es un tipo particular de variable capaz de almacenar *las direcciones de memoria* de otras variables y, a través de diversos operadores, acceder directamente a los datos en esas direcciones de memoria y modificar su contenido. De esta manera los punteros dan un camino secundario para operar sobre los datos de las variables sin necesidad de utilizar su nombre.

Los punteros son herramientas muy importantes del lenguaje C, pero su utilidad es difícil de comprender antes de ver temas más avanzados como memoria dinámica y estructuras de datos enlazadas, porque la mayoría de las operaciones sencillas que pueden realizarse con punteros también pueden realizarse sin ellos. Sin embargo, con un poco de paciencia, al final de este mismo capítulo veremos el primer uso fundamental de los punteros: el pasaje de información por referencia a las funciones.

Direcciones de memoria

Como se mencionó en el capítulo 1, la mínima unidad de memoria capaz de retener información es el bit, que puede almacenar dos valores (0 o 1). Un conjunto de 8 bits se denomina byte; la memoria de la computadora suele organizarse como una serie de bytes donde nuestros programas de C almacenarán las variables que utilizan. Cuando escribimos un programa declaramos variables, con lo cual se "reserva" espacio en la memoria para alojarlas; se reserva tanto espacio como necesite la variable según su tipo. Al momento de declarar variables les asignamos nombres (una especie de etiqueta) y utilizamos esos nombres para acceder a los valores almacenados en esas variables, leerlos y modificarlos. Veamos el ejemplo del código 6.1.

```
int apellido[10] = "Ferrer";
int fila = 5;
char butaca = 'H';
printf("%s: %d-%c", apellido, fila, butaca);
```

Código 6.1. *Programa de demostración de posiciones de memoria asociado a la figura 1.*

Al ejecutarse este programa, los valores asignados a las variables se guardan en la memoria de acuerdo al sistema de representación correspondiente. Evidentemente, la computadora puede acceder nuevamente a la información cuando la necesita, por ejemplo en la línea 4. El mecanismo interno por el cual puede hacerlo es a través de las direcciones de memoria asignadas a las variables. La figura 6.1 muestra cómo podría verse un posible mapa de la memoria de la computadora, donde hemos asignado un valor arbitrario de 2000 al comienzo de las direcciones de memoria.

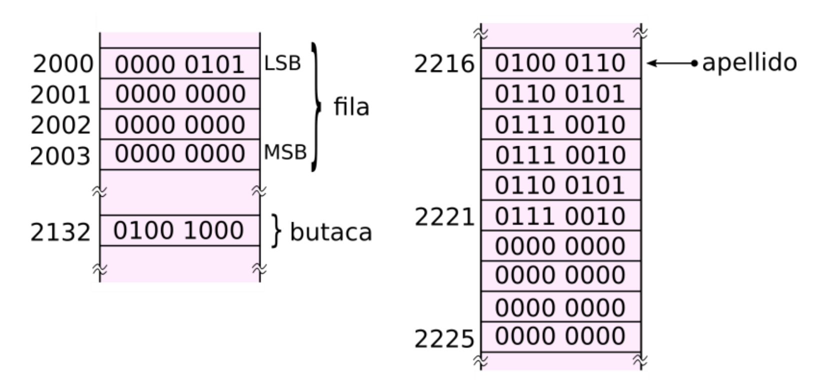


Figura 6.1. *Tabla de memoria para las variables del código 6.1.* `fila` *es una variable de tipo* `int` *de 4 bytes, butaca es de tipo* `char` *y* `apellido` *es un arreglo de 10 elementos*

Las variables tienen una dirección de inicio, un tamaño y un nombre o etiqueta asignado para manipular ese espacio de memoria. Las variables, como `fila` y `butaca` en el ejemplo, en general son independientes unas de otra y son colocadas en direcciones de memoria sobre las que, en principio, no podemos suponer nada: no hay reglas que dicten si serán posiciones contiguas o no. En cambio, los *arreglos* tienen la particularidad de ser almacenados a partir de una cierta posición con todos sus elementos seguidos.

Para poder leer una variable de la memoria de la computadora entonces es necesario conocer la dirección de inicio, y el tamaño de la misma. Por supuesto, también debe saberse el sentido en el que se almacenan sus bytes (¿hacia adelante o hacia atrás?), y si el primer byte es el más significativo o el menos significativo.

Una variable puede comenzar en una posición y sus bytes almacenarse en posiciones crecientes o decrecientes; sin embargo esto es en general una característica de bajo nivel del sistema de cómputo y como programadores de C no nos preocuparemos por eso. A los fines de las ilustraciones en este capítulo, asumiremos que los bytes se organizan en posiciones crecientes de memoria comenzando por el byte menos significativo de la variable (LSB por sus siglas en inglés), hasta el más significativo (MSB) como se ilustra en la misma figura 1.

Afortunadamente, a la hora de acceder a las variables por su dirección de memoria, necesidad que surge en programas complejos de C, quien programa no debe lidiar con todas estas características de bajo nivel sino que declara y utiliza sencillamente punteros.

Punteros

Declaración

Un puntero se declara explicitando un tipo, el símbolo `*` y un nombre, de la siguiente manera:

tipo * nombre;

En el código 6.2 se muestran ejemplos de la declaración de punteros.

```
int *p; /* Puntero de tipo int con nombre p */
char *car1, *car2;   /* Dos punteros de tipo char, nombres car1 y car2 */
double *arrp[10];  /* Un arreglo de 10 punteros de tipo double */
float f1, *pf;   /* Una variable de tipo float y otra puntero de tipo float */
```

***Código 6.2.** Ejemplo de declaración de punteros.*

En las declaraciones del código 6.2, resalta el uso del símbolo `*` inmediatamente detrás del nombre de lo que se quiere declarar como puntero. Se suele decir para un puntero de cierto tipo, por ejemplo `int`, que es un "puntero a entero" porque puede apuntar a una variable de ese tipo. Otro ejemplo: "Puntero a `char`" es equivalente a decir "puntero de tipo `char`".

Operadores * y &

Un puntero sirve para modificar el valor de una variable, pero para poder hacerlo es necesario primero asignarle la dirección de memoria de esa variable. Eso puede lograrse con el operador & (llamado *ampersand*[15]), que devuelve la dirección de memoria de una variable.

Si a es una variable cualquiera, el operador & devuelve la posición de memoria o dirección de a a través de la expresión &a. Esa dirección puede almacenarse en un puntero del mismo tipo que a.

En las líneas 1 a 3 del código 6.3 se demuestra el operador. Se declara un puntero a entero de nombre `p1` y un entero `a`. Luego se obtiene la dirección de memoria de `a` utilizando la expresión `&a` y se almacena en `p1`, que justamente está preparado para guardar direcciones de memoria de enteros.

[15] El operador & se llama "*ampersand"* en inglés y "et" en castellano, pero en nuestro idioma es muy poco utilizado por lo que nos referiremos a él por su nombre más conocido en inglés.

Una vez que se tiene una dirección de memoria en un puntero, puede leerse y también modificarse el contenido de esa posición utilizando el puntero y el operador `*`, llamado operador de *desreferenciación* o de *indirección*.

Si `p` es un puntero, la expresión `*p` permite acceder al contenido de la dirección de memoria almacenada en `p`, es decir, al contenido de la variable a la que apunta `p`.

En la línea 4 del código 6.3 continúa el ejemplo mostrando cómo el operador de desreferenciación permite modificar el contenido de una variable cuya dirección se guardó en el puntero `p`, y en la línea 6 se muestra que de la misma manera puede leerse el contenido.

```
1  int a = 0; /* Declaración de un entero de nombre a inicializado en 0 */
2  int *p1; /*Declaración de un puntero a entero de nombre p1 */
3  p1 = &a; /* Obtiene la dirección de a con el operador & y se almacena en p1 */
4  *p1 = 5; /* Se modifica el contenido de a utilizando p1 y el operador *  */
5  printf("%d\n", a);   /* Se imprime 5 */
6  printf("%d\n", *p1); /* Se imprime 5 */
```

Código 6.3. *Uso básico de un puntero con los operadores & y *.*

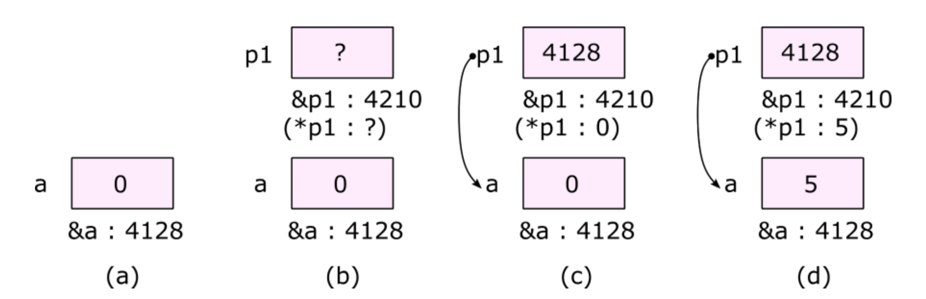


Figura 6.2. *Representación gráfica de la memoria al ejecutarse el Código 6.3 tras ejecutar las líneas 1, 2, 3, y 4 en las columnas (a), (b), (c) y (d) respectivamente.*

En la figura 6.2 se muestra gráficamente la ejecución del Código 6.3. Cada caja representa la memoria donde está guardada la variable (que en realidad se compone de varios bytes, según el tipo). El nombre de la variable está anotado a la izquierda de la caja, y su posición de memoria debajo. Como ya hemos visto, el operador `&` devuelve la posición de memoria por lo que usaremos `&a` para abreviar “la posición de memoria de `a`”. Por supuesto, el puntero `p1` es en sí mismo una variable y tiene por tanto su propia dirección de memoria. También se grafica debajo de la “caja” correspondiente a `p1` el resultado de aplicarle el operador `*`, que es el valor de la variable a la que apunta `p1`. A lo largo de los pasos (a), (b) y (c) puede apreciarse cómo se asigna la posición de memoria de `a` al puntero `p1` y luego se lo utiliza para modificar el contenido de la variable `a`.

Los punteros son variables y como tales pueden almacenar distintos valores en cada momento, como se ve en el Código 6.4.

```
double f1=2.5,f2=4;
double prod = 1;
double *pf = &f1;
prod = prod * (*pf);
pf = &f2;
prod = prod * (*pf); /* prod vale 10 luego de la línea 6 */
```

Código 6.4. Ejemplo de multiplicación con punteros

Y de la misma manera que otras variables, pueden también transferirse sus valores si son del mismo tipo (o con la conversión explícita correcta si son de tipos distintos), como se ve en el Código 6.5, donde se debe prestar especial atención a la línea 6.

```
1 double v = 2.5;
2 double d = 0;
3 double *pd, *pv1,*pv2;
4 pd = &d;
5 pv1 = &v;
6 pv2 = pv1;
7 *pd = *pv1 + *pv2; /* d vale 5 luego de la línea 7 */
```

Código 6.5. Ejemplo de duplicación de un valor con punteros

En este caso, la dirección de memoria de `v` se almacena en `pv1` (línea 5), y luego se copia también en `pv2` (línea 6).

Debe notarse que, como sucede con cualquier otra variable, al momento de declarar un puntero no podemos saber qué contiene si no se inicializó. Sin embargo, mientras que una variable numérica que contiene valores espurios es relativamente inofensiva, el caso de los punteros es particularmente problemático ya que si se aplica el operador `*`, se interpretará el contenido del puntero como una dirección de memoria y el programa intentará acceder a una posición de memoria por fuera de su rango accesible. Si el programa corre en un sistema operativo, probablemente el sistema termine la ejecución del mismo debido a este comportamiento.

Finalmente, aunque pueda resultar trivial, es valioso comprender que podemos aplicar el operador de desreferenciación a cualquier dirección de memoria aunque no esté almacenada en un puntero. Así, la expresión `*(&x)` resulta igual a `x`, ya que es el contenido de la variable que está en la dirección de `x`, que es justamente `x`. Nunca escribiríamos algo así ya que carece de sentido práctico, pero puede ser un paso intermedio al resolver expresiones complicadas.

Aritmética de punteros

Los arreglos tienen la particularidad de estar organizados en memoria secuencialmente. Por ejemplo, en un arreglo de tipo `char` (cada `char` ocupa 1 byte) sabemos que si un elemento se encuentra en la dirección n, el siguiente elemento se encontrará en la dirección n+1, el próximo

en n+2, y así. Si el arreglo es de tipo `int` de 4 bytes y el primer elemento está en la posición n, el siguiente elemento se encontrará en la posición n+4, el próximo en n+8, etc.

Por esta propiedad de contigüidad en memoria, los punteros resultan muy útiles y naturales para trabajar con arreglos ya que pueden “apuntarse” a una dirección del arreglo y luego “desplazarse” a lo largo del arreglo. Ahora bien, si para cada tipo de dato fuese necesario hacer la cuenta de su tamaño y sumar ese tamaño multiplicado por la cantidad de posiciones que se quiera desplazar, el código sería engorroso y proclive a errores. Por lo tanto, el lenguaje C incluye la característica conocida como “aritmética de punteros”: Si se suma un entero a una variable puntero de tipo T, la dirección de memoria que contienen el puntero se incrementará en tantos bytes como el tamaño del tipo T

En otras palabras, si se *incrementa en uno* un puntero de *un cierto tipo*, el puntero apuntará al siguiente elemento *del mismo tipo*; el compilador se encarga automáticamente de sumar el número correcto a la dirección de memoria para incrementar adecuadamente el puntero. Esta característica se ejemplifica en el Código 6.6 y la Figura 6.3.

```
short int arr1[4] = {1,2,3,4};
int arr2[4] = {1,2,3,4};
short int *p1;
int *p2;
p1 = &arr1[0];
p2 = &arr2[0];
printf("%d\n",*p1); /* 1 */
printf("%d\n",*p2); /* 1 */
p1 = p1 + 1;
p2 = p2 + 1;
printf("%d\n",*p1); /* 2 */
printf("%d\n",*p2); /* 2 */
```

Código 6.6. *Ejemplo de aritmética de punteros.*

Figura 6.3. *Resultado de sumar 1 a un puntero de tipo `short int` de 2 bytes (`p1`) y a uno de tipo `int` de 4 bytes (`p2`) como en el código 6.6.*

En el Código 6.6 se declara un arreglo `arr1` de tipo `short int` y se apunta un puntero `p1` al comienzo del mismo. Asumimos un sistema donde el tipo `short int` ocupa 2 bytes. A su vez, se declara `arr2` de tipo `int` (que asumiremos ocupa 4 bytes) y se apunta un puntero `p2` de tipo `int` a su inicio. Finalmente, se incrementan ambos punteros en 1 pero el efecto es distinto para cada uno ya que tienen distintos tipos: mientras `p1` se incrementa 2 bytes, `p2` se incrementa en 4.

Todas las operaciones matemáticas de adición y sustracción que se realizan con enteros pueden aplicarse sobre los punteros. En particular es muy usual utilizar los operadores de post-incremento y pre-incremento. Si `p` es un puntero `p++` y `++p` pueden usarse para incrementar su posición de memoria al siguiente elemento (recordando la diferencia entre pre y post incremento vista en el capítulo 3) y `p--` o `--p` para decrementarla al anterior.

Es muy común ver programas en C que combinan estos operadores con el de desreferenciación. Esto puede resultar confuso y debe realizarse con cuidado, recordando el orden de precedencia de operadores. Analicemos las líneas del Código 6.7 siguiendo la traza de la tabla 6.1.

```
1  vars[]={10,20,30,40},
2  int *punt;
3  punt = &vars[0];
4  punt++;
5  printf("%d\n",*punt);
6  printf("%d\n",*punt++);
7  printf("%d\n",(*punt)++);
8  printf("%d\n",*punt);
```

Código 6.7: *Operadores aritméticos sobre punteros*

Tabla 6.1. Traza del programa del código 6.7

Linea	punt	vars				Salida por pantalla
		vars[0]	vars[1]	vars[2]	vars[3]	
1		10	20	30	40	
2	?	10	20	30	40	
3	&vars[0]	10	20	30	40	
4	&vars[1]	10	20	30	40	
5	&vars[1]	10	20	30	40	20
6	&vars[2]	10	20	30	40	20
7	&vars[2]	10	20	31	40	30
8	&vars[2]	10	20	31	40	31

En las líneas 1 y 2, se declara un arreglo `vars` de enteros (inicializado con valores) y un puntero `punt` a entero que, en la línea 3, se “apunta” a la dirección del primer elemento del arreglo, `vars[0]`. Luego, en la línea 4 se aplica la operación de postincremento a `punt` por lo cual su dirección de memoria se incrementa hasta apuntar al siguiente entero, `vars[1]`. En la línea 5 se utiliza el operador de desreferencia para acceder *al contenido de la variable a la que apunta* punt. Como punt apunta a `vars[1]`, ese contenido es 20 y eso es lo que se le pasa a la función `printf()` para imprimir en pantalla.

La línea 6 exhibe un comportamiento que exige un análisis más detallado. Para saber qué se imprime en pantalla debemos analizar la expresión `*punt++`. Primero debe notarse que hay *dos* operadores, `*` y `++`, operando sobre el puntero. La precedencia es: primero `++` y luego `*`, por lo que la operación se lee como `*(punt++)`; es decir que lo primero que debemos evaluar es `punt++` y luego aplicar el `*` a lo que resulte esa expresión. Pero debe notarse algo más antes de continuar: `++` está *a la derecha* de `punt` y por lo tanto es el operador de *post*incremento: siempre se ejecuta *luego* de terminar de evaluar la expresión completa. Para pensarlo de otra manera, el postincremento incrementa la variable pero “devuelve” el valor anterior al incremento.

Por lo tanto, en la expresión `*(punt++)`, reemplazamos `punt++` por `&vars[1]`. Entonces, `*(&vars[1])` recupera el valor ubicado en la dirección de memoria `&vars[1]` que resulta 20. Al mismo tiempo, `punt` pasa a valer `&vars[2]`, y debemos recordarlo al terminar de analizar la expresión para anotar el valor correcto en la traza.

La siguiente línea, la línea 7, agrupa los mismos operadores que en la línea 6 pero de otra manera: `(*punt)++` primero desreferencia `punt`, que apunta a `&vars[2]` y por lo tanto contiene el valor 30. A continuación, aplica el postincremento sobre ese valor (no sobre el puntero), nos “devuelve” 30 para usar en el resto de la expresión (la llamada a `printf()`) y luego lo incrementa a 31. Finalmente en la última línea se imprime el valor de la variable a la que apunta `punt`, que sigue siendo `vars[2]` y vale 31.

Arreglos y punteros

Equivalencia entre arreglos y punteros

Los arreglos y los punteros están estrechamente relacionados en C, incluso más allá de la aritmética de punteros que permite recorrer los arreglos fácilmente. Al momento de acceder a uno de sus elementos, el arreglo es convertido automáticamente a un puntero que contiene la dirección de memoria del primer elemento. Es decir que los nombres de los arreglos pueden verse como punteros, pero constantes, ya que no pueden modificarse una vez declarados.

La declaración

```
int datos[10]
```

reserva memoria para un arreglo de 10 elementos de nombre “datos” y en nuestro programa podremos usar `datos` como un puntero constante al inicio del arreglo, es decir que si `p` es un puntero a entero, las siguientes expresiones son equivalentes:

```
p = &datos[0]
p = datos
```

Reiterando la idea previa, notemos nuevamente que `datos` no puede modificarse y por lo tanto mientras una expresión como `p = &x` es válida si `x` es un entero, nunca podríamos escribir

```
datos = &x /* Expresión inválida */
```

o `datos++ /* Otra expresión inválida*/`

Incluso, el operador de indización `[ ]` se define formalmente considerando que la expresión `x[i]` debe interpretarse como `*(x+i)`, expresión que sabemos analizar gracias al estudio de punteros en las secciones previas.

Distintas formas de manipular arreglos

El nombre de los arreglos puede usarse como referencia para acceder a sus elementos, como en el ejemplo del código 6.8 cuya salida por pantalla se muestra en la figura 6.4.

```
1  int datos[5] = {10,20,30,40,50};
2  int i;
3  int *p = datos;
4  for(i=0;i<5;i++){
5      printf("%d ",datos[i]);
6      printf("%d ",*p++);
7      printf("%d\n",*(datos+i));
8  }
```

Código 6.8. *Equivalencia entre el uso de índices, un puntero, y el nombre del arreglo.*

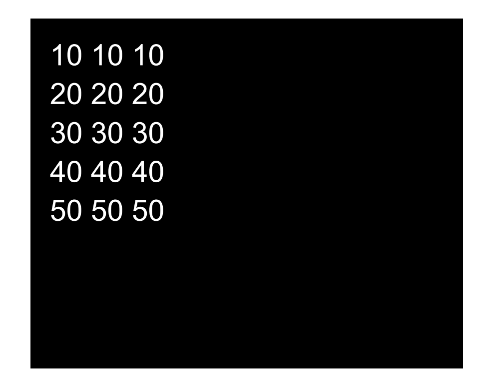


Figura 6.4. *Salida por pantalla del código 6.8.*

Puede verse en el código 6.8, línea 3, que la dirección de memoria que representa `datos` (que es un arreglo de enteros) puede almacenarse en `p`, que es un puntero a entero.

En la línea 7 se observa la expresión `*(datos+i)` que se resuelve comenzando por aplicar aritmética de punteros a `datos+i`. Recordemos que C convierte `datos` a un puntero a la primer dirección del arreglo, es decir que vale `&datos[0]`. Al sumarle un número entero `i`, se obtiene la posición de memoria del valor que está a `i` enteros de ese lugar, o, lo que es lo mismo, `&datos[i]`. Luego al aplicar `*(&datos[i])` se tiene como resultado `datos[i]`.

Arreglos multidimensionales

Cuando se trata de arreglos bidimensionales, ocurre algo conceptualmente similar a los arreglos unidimensionales, pero más complejo. Consideremos un arreglo de 3 filas y 4 columnas de tipo `int`:

```
int mat[3][4] = {  { 1, 2, 3, 4},
                   { 5, 6, 7, 8},
                   {9, 10, 11, 12} };
```

Al declararlo se reserva espacio en memoria para ubicar a los elementos de la matriz en forma contigua y secuencial, colocando una fila después de la otra como grafica la figura 6.5.

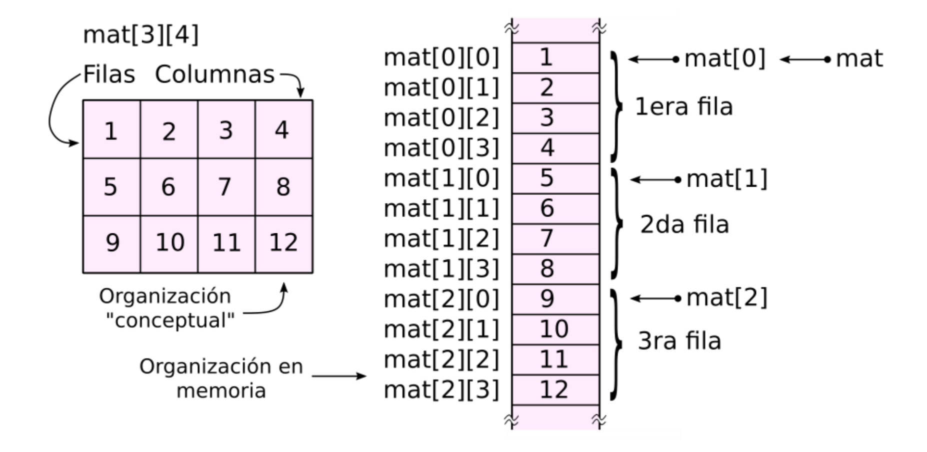


Figura 6.5. *Representación de un arreglo bidimensional como matriz (izquierda) y según su almacenamiento en memoria (derecha). Se señalan los punteros `mat[i]` y `mat`.*

La matriz se organiza por lo tanto como una colección de filas, o “arreglo de filas”, y por lo tanto `mat[0]`, `mat[1]` y `mat[2]` representan las direcciones de memoria de cada una de las filas.

Entonces, dada por ejemplo la declaración

`<tipo> mat[N][M]`

cuando queremos acceder al elemento `i`,`j` de la matriz escribiendo `mat[i][j]`, podemos interpretar que se accede a la fila i-ésima a través de `mat[i]` o `*(mat+i)` lo cual devuelve la posición de memoria del arreglo de M elementos que constituye la fila i, y luego se accede al elemento particular aplicando nuevamente el índice con `(*(mat+1))[j]`. Si encadenamos las reglas anteriores, esta última expresión puede escribirse como `*(*(mat+i)+j)`.

Si bien `mat` es un tipo de dato “arreglo bidimensional”, se observa en la Figura 6.5 que en realidad la matriz en memoria está organizada como un gran arreglo sencillo con una fila tras otra y podríamos recorrerlo de una punta a la otra con un puntero como en el código 6.9. En el código 6.9 se demuestra el uso de un puntero `p` a entero para recorrer los elementos de un arreglo bidimensional de enteros. El puntero se apunta a la dirección de inicio de la matriz, utilizando para ello una operación de *cast* ya que su tipo (arreglo bidimensional de enteros) no es el mismo que el de `p` (puntero a entero).

```c
#define N 2
#define M 3
int main()
{
    int mat[N][M] = {1,2,3,4,5,6};
    int i,j;
    int *p = (int*)mat;
    for(i=0; i<N; i++){
        for(j=0; j<M; j++){
            printf("%d ",mat[i][j]);
            printf("%d\n", *p++);
        }
    }
    return 0;
}
```

Código 6.9. *Dos formas de recorrer un arreglo bidimensional.*

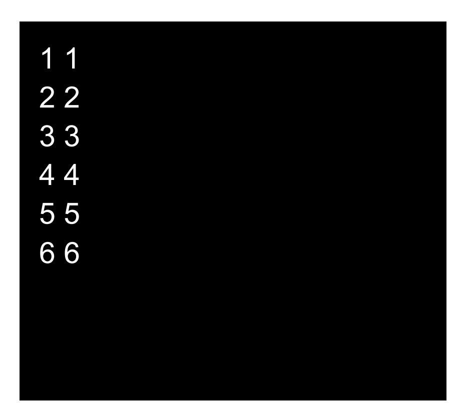


Figura 6.5. *salida por pantalla del código 6.9.*

En secciones anteriores aprendimos a manipular los datos de un arreglo unidimensional utilizando punteros, y para ello necesitábamos apuntar el puntero al arreglo, lo cual se lograba con líneas como las siguientes:

```c
int *p;
int arr[10];
p=arr;
```

Si se quiere manipular análogamente un arreglo bidimensional, la situación es más compleja ya que el siguiente código, que sería natural en principio escribir, es erróneo:

```
int *p;
int x[3][5];
p=x; /* CÓDIGO ERRÓNEO */
```

Recordemos que al aplicar el primer índice de la matriz, `x[i]` en realidad devuelve la dirección de memoria del primer *arreglo de 5 elementos*, es decir, la primer "fila", por lo tanto su tipo equivalente sería el de "puntero a arreglo de 5 enteros". Con esta información podemos corregir el código anterior:

```
int (*p)[5];
int x[3][5];
p=x; /* CÓDIGO CORRECTO */
```

Y aún más, obtenemos que las siguientes líneas son equivalentes:

```
x[i][j];
*(*(x+i)+j);
*(*(p+i)+j);
p[i][j];
```

Pasaje de parámetros por referencia

En este punto llegamos al primer tema donde el uso de punteros es fundamental y demuestra una tarea que no puede lograrse de otra manera.

En un capítulo previo se trató el tema de funciones, una manera de crear "módulos" con nombre en nuestros programas que cumplen una misión determinada. Analicemos el ejemplo del código 6.10 a continuación.

```
int diventera(int, int);
int main(){
    int n1=5, n2=2;
    int r = diventera(n1,n2);
    printf("%d",r);
    return 0;
}
int diventera(int a, int b){
    int res;
    res = a/b;
    return res;
}
```

Código 6.10. *Función de división entera.*

En este ejemplo se declara una función que toma dos argumentos enteros y devuelve un entero resultado de dividir el primer argumento por el segundo. Los números para realizar la operación se pasan *por valor*, es decir, que al llamar a la función `diventera()` los valores contenidos en `n1` y `n2` (en este caso 5 y 2) se transfieren a las variables en el argumento de la función, en este caso `a` y `b`. Luego de realizar la operación de división, la función devuelve el valor resultante al programa principal a través de su valor de retorno, el cual es recibido en la variable `r`.

Hay algunos problemas con esta función: si se le pasa un 0 como segundo argumento, intentará dividir por 0 y ocurrirá un error en el programa. Pero incluso aunque se comprobase si el valor es cero, no habría manera de comunicar al programa principal que no se puede realizar la operación, ya que sólo se devuelve un único valor que corresponde al resultado. Sería deseable poder devolver dos valores al programa principal, uno donde se devuelva el resultado, y otro para indicar si la operación se realizó correctamente.

Cuando se quieren devolver *varios* valores desde una función al código que la llama, se utiliza el método de *pasaje por referencia*. En este método se declara una variable como `r` para recibir información y se le comunica a la función *la dirección de memoria* de esa variable, para que la función pueda guardar allí el resultado. Al terminar la función, la información estará almacenada en una variable disponible en el ámbito desde donde se la llamó.

Los punteros son fundamentales para poder manipular la dirección de memoria que hay que comunicar a la función. Veamos una versión mejorada de la función del código 6.10 en el código 6.11.

```
int diventera(int, int, int *);
int main(){
    int n1=5, n2=2;
    int r;
    int e = diventera(n1,n2,&r);
    if(e==1){
        printf("Ha ocurrido un error");
    }
    else{
        printf("%d",r);
    }
    return 0;
}
int diventera(int a, int b, int *res){
    if(b==0){
        return 1;
    }
    else{
        *res = a/b;
        return 0;
    }
}
```

Código 6.11. *Ejemplo ilustrativo de pasaje por referencia.*

En contraste con el código 6.10, vemos que la función tiene un argumento más, de tipo puntero ya que su utilidad será recibir una dirección de memoria. El programa espera un valor de retorno por parte de la función que indica si hubo algún problema (en cuyo caso retorna 1) o si todo salió bien, en cuyo caso retorna 0. La función utiliza su valor de retorno para devolver esta información al programa principal, y por lo tanto debe encontrar otro camino para retornar el resultado de la división. Ese camino es provisto por el parámetro `res` que admite una dirección de memoria. La función debe almacenar el resultado en la dirección de memoria que le pasen, y

por eso en la línea 19 se utiliza la expresión `*res = a/b` que permite modificar el contenido de la variable a la que apunta `res`. `res` tiene el valor `&r`, y por lo tanto `r` toma el valor `a/b`.

Esta forma de organizar las funciones parece extraña en principio pero es usual en la práctica, y resulta consistente si consideramos que las funciones pueden tener que devolver varios valores pero alcanza con un sólo entero para retornar distintos códigos de error.

Veamos un segundo ejemplo sencillo en el código 6.12. En este código se observa una función de tipo `void`, es decir que no retorna nada, pero aún así puede “transmitir” al programa principal el resultado de la operación que realiza a través del pasaje por referencia, modificando directamente la variable del `main()` gracias a conocer su dirección de memoria.

```
void duplicar(float *);
int main(){
    int r=23.2;
    duplica(&r);
    printf("%d",r); /* Imprime 46.4 */
    return 0;
}
void duplicar(float *a){
    *a = 2*(*a);
}
```

Código 6.12. *Función de tipo `void` que devuelve información a través del pasaje por referencia.*

El pasaje por referencia es fundamental a la hora de pasar arreglos a las funciones. Este tema se abordó en el capítulo sobre funciones con cierto “maquillaje”, ya que C permite incluir sintaxis de arreglos en los argumentos de las funciones. Sin embargo, lo que realmente sucede “tras bambalinas” es que a la función se le pasa la dirección de memoria del primer elemento del arreglo. En el código 6.13 se programa una función a la que se le pasa una cadena y retorna el largo.

```
#define MAX 50
int largo(char *, int max);
int main(){
    char cad[MAX] = "hola mundo";
    int r;
    r = largo(cad,MAX);
    printf("%d",r);
    return 0;
}
int largo(char * pcad, int max){
    int n=0;
    while(*pcad != '\0' && n < max){
        pcad++;
        n++;
    }
    if(n==max)
        return -1;
    else
        return n;
}
```

Código 6.13. *Pasaje de arreglos.*

La función `largo` del código 6.13 tiene como argumento un puntero a `char`. Este puntero le permite recibir la dirección de memoria de cualquier variable de tipo `char`, por ejemplo el primer elemento de un arreglo. Al pasarle un arreglo que contiene una cadena, el puntero puede ser incrementado dentro de la función hasta que la variable a la que apunta contenga el caracter de terminación de cadena.

Los punteros crean un "túnel" que permite manipular a las variables sin usar la "puerta principal" que es su nombre. Este túnel puede ser usado de la manera más conveniente, y resulta fundamental para establecer caminos de comunicación "por debajo" de las murallas que establecen las funciones y sus ámbitos locales de variables. Parecería una buena analogía entonces el túnel de un ladrón que quiere robar un banco. Sin embargo, estos túneles excavados con punteros no sólo nos dan acceso a las pequeñas bóvedas de las variables, sino que una vez atravesados nos permiten desplazarnos, gracias a la aritmética de punteros, a las direcciones aledañas y más allá. Es tentador cambiar la analogía por la de un verdadero túnel que une dos ciudades por debajo de un río, pero no debemos engañarnos: la realidad se asemeja más al túnel de un prisionero escapando. Si se trata del héroe bienintencionado e incorrectamente aprisionado, el túnel da ocasión para el festejo, si en cambio se trata del villano malhechor, las consecuencias pueden ser catastróficas. El puntero nos da la posibilidad de modificar cualquier valor de la memoria sin limitaciones de ámbito o tipo, y nada controla que no nos excedamos del rango de memoria "seguro" de nuestro programa. Por este motivo, los punteros pueden considerarse una característica "insegura" de C y lenguajes posteriores de más alto nivel los reemplazan por otras herramientas. Sin embargo, en C los punteros son una herramienta fundamental y pueden usarse con muy buenos resultados; sólo debe recordarse la frase popularizada por el tío del famoso superhéroe de los cómics: "un gran poder conlleva una gran responsabilidad".

Ejercicios

1) Realice un programa donde el usuario ingrese por teclado los valores de dos variables (un entero y un carácter), los incremente en 1 utilizando punteros e imprima dichas variables en pantalla.

2) Analice, ejecute y verifique el siguiente código.

```
#include <stdio.h>
int j, k;
int *ptr;
int main()
{
    j = 1;
    k = 2;
    ptr = &k;
```

```
    printf("\n");
    printf("j tiene el valor %d y está almacenado en %p\n",j,&j);
    printf("k tiene el valor %d y está almacenado en %p\n",k,&k);
    printf("ptr tiene el valor %p y está almacenado en %p\n",ptr,&ptr);
    printf("El valor del entero al que apunta ptr is %d\n", *ptr);
    return 0;
}
```

3) Dado el siguiente fragmento de código

```
    float a = 23.56;
    float *b;
    float *c;
    b = &a;
    c = b;
    a = *c + *b;
```

a) Indique las afirmaciones correctas
   i) Las variables `b` y `c` se almacenan en la misma dirección de memoria
   ii) Si se incluyera la sentencia `*c = 4;`, modificaría el contenido de a.
   iii) `a` tomará un valor indeterminado.
   iv) `c` almacena la dirección de la variable `b`.

b) ¿Cuál es el valor final de la variable `a`?

4) Dado el siguiente fragmento de código:

```
int v[3] = {1,2,3};
int *p;
int r;
p = v;
```

Explique las similitudes y diferencia entre las siguientes líneas:

```
a)  r = v[1];
b)  r = *(v+1);
c)  r = *(p+1);
d)  p = p+1;
r = *p;
```

5) Realice un programa que defina un arreglo formado por los enteros {2, 10, 9, 4, 90, 56}, defina un puntero ptr al primer elemento del arreglo, y recorra y muestre el valor de cada elemento del arreglo utilizando índices y, luego, utilizando el puntero ptr.

6) Escribir un programa que calcule y visualice la media aritmética de un vector de 10 elementos numéricos enteros entrados por teclado, utilizando una variable puntero a dicho vector.

7) Analice el siguiente código:

```
#include <stdio.h>
char strA[40] = "Solamente una cadena de prueba";
char strB[40];
int main(){
   char *pA;
   char *pB;
   puts(strA);
   pA = strA;
   puts(pA);
   pB = strB;
   putchar('\n');
   while(*pA != '\0'){
       *pB++ = *pA++;
   }
   *pB = '\0';
   puts(strB);
   return 0;
}
```

a) Ejecute el programa, utilice el debugger en su opción paso a paso, mire las variables `strA`, `strB`, `pA` y `pB`.

b) Inicialice `strB[40]` con la siguiente cadena y saque conclusiones:

“012345678901233456789012345678901234567890123456789 0”

c) Reemplace el código necesario con la función de librería de C `strcpy()`.

8) Realice una función de nombre `Calcular( )` que tenga como primer argumento un arreglo de elementos tipo `double` y como segundo el tamaño del mismo. Complete los parámetros y el valor de retorno para obtener de la función: la media aritmética, la cantidad de veces que se repite el máximo, y la cantidad de veces que se repite el mínimo en el arreglo.

9) Indique la salida por pantalla de los siguientes códigos

```
int x;
void Calculo(int *n);
int main(){
  x = 20;
  Calculo(&x);
  printf("x = %d\n", x);
  return 0;
}
void Calculo(int *n){
  x = x + 5;
  *n = *n + 3;
  return;
}
```

```
int x, y;
void fun1(int *x, int y);
int main(){
  x = 13; y = 12;
  fun1(&x, y);
  fun1(&y, x);
  printf("x = %d\n", x);
  printf("y = %d\n", y);
}
void fun1(int *x, int y){
  y = *x;
  *x = y;
}
```

```
int x, y;
void funx(int *x, int *y);
int main( ){
  x = 1; y = 2;
  funx(&y, &x);
  printf("x = %d\n", x);
  printf("y = %d\n", y);
}
void funx(int *x, int *y){
  *x = *y + 2;
  *y = *x + 2;
}
```

10) Implemente las siguientes funciones y programas para demostrar su uso:

a) Una función de nombre `cadlen()` que retorne la cantidad de caracteres presentes en una cadena pasada como argumento.

b) Una función de nombre `cadcat()`, que tenga como primer argumento la cadena destino y como segundo la cadena origen a añadir al final de la cadena destino.

11) La siguiente función imprime de manera iterativa una cadena de caracteres:

```
void imprimir_iter(char * pcad){
    while(*pcad != '\0'){
        printf("%c",*pcad);
        pcad++;
    }
}
```

a) Realice la versión recursiva de la función anterior.

b) Modifique la función recursiva para que imprima la cadena invertida.

12) Hacer una función para conocer el número de veces que aparecen cada dígito decimal en un determinado número natural. El primer argumento de la función será el valor del número a analizar de tipo `unsigned long int` y el segundo será un arreglo de 10 elementos del tipo `unsigned int` donde se guardarán las ocurrencias correspondientes a cada dígito. Por ejemplo, para el número 248282, la función deberá devolver todo el array en cero, excepto la posición 2 que tendrá un 3, la posición 4 que tendrá un 1 y la posición 8 que tendrá un 2.

13) Seguir la traza de los siguientes programas manualmente y determinar que imprimen en pantalla. Verifique el resultado utilizando el depurador o *debugger* del IDE

```
int a, b;
void E2(int a, int b);
int main()
{
  a = 1;
  b = 2;
  E2(b, a);
  printf("%d %d\n",a,b);
  return 0;
}

void E2(int a, int b)
{
  a = 10;
  printf("%d %d\n",a,b);
  return;
}
```

```
int y, z;
void Cambiar(int *i, int j);
int main()
{
 y = 21;
 z = 7;
 Cambiar(&y, z);
 Cambiar(&z, y);
 printf("%z=%d y=%d\n",z,y);
 return 0;
}

void Cambiar(int *i, int j)
{
 int k;
 k = j;
 6 *i = k + j;
 k = *i;
 return;
}
```

CAPÍTULO 7
Estructuras

Pablo A. García

Introducción

La forma más sencilla de comprender qué son las estructuras consiste en identificarlas como una entidad que reúne un conjunto de variables relacionadas, y que se utiliza para modelar algún objeto de interés en nuestro código. También suelen denominarselas como agregados de datos. Por ejemplo, un programador podría modelar un alumno haciendo uso de una estructura. Del alumno a modelar es de interés: el nombre (una cadena), el apellido (una cadena), el legajo (un unsigned int), el promedio (un real), el año de ingreso (un entero) y la cantidad de materias aprobadas (un entero). En el siguiente punto se presenta la forma de declarar y definir este nuevo tipo de dato (Estructura alumno) haciendo uso de estructuras.

Las estructuras pueden contener variables de distintos tipos, a diferencia de los arreglos, que contienen elementos de un único tipo.

Estas son muy utilizadas para la generación de listas de datos modelados mediante estructuras, que en conjunto con el uso de archivos binarios, simplifica el almacenamiento de la información.

Por otro lado, la inclusión de punteros en las estructuras, permite generar “estructuras autoreferenciadas” que facilitan el modelado de estructuras de datos más complejos como son las pilas, las colas y las listas.

Definición de estructuras

Para nuestro modelo de alumno antes descrito, consideremos la siguiente definición de estructura:

```
struct alumno{
      char nombre[30];
      char apellido[30];
      unsigned int legajo;
      float promedio;
      unsigned int ingreso
      unsigned int cantidad_materias;
               };
```

En esta declaración, el uso de la palabra reservada `struct` presenta la definición de la estructura. El identificador alumno es el nombre de la estructura, y utilizaremos el conjunto `struct alumno` para declarar variables de este nuevo tipo de dato que estamos generando. Las variables declaradas dentro de las llaves son los miembros de la estructura (nombre, apellido, legajo, promedio, ingreso y cantidad_materias). En este ejemplo, la estructura está formada por tipos básicos y cadenas de caracteres, pero también pueden incluirse otras estructuras como miembros de estructuras, formando las denominadas estructuras anidadas (se describen más adelante en este capítulo).

La definición de la estructura que se ha presentado no reserva espacio en memoria para ninguna variable, tan solo genera un tipo nuevo de dato que se utilizará para declarar variables. Por ejemplo, la siguiente declaración:

```
struct alumno al1, Alumnos[100],*pal;
```

está declarando la variable `al1` como una variable del tipo `struct alumno`, `Alumnos` como un arreglo de 100 elementos del tipo `struct alumno` y `pal` como un puntero a `struct alumno`.

Estas tres mismas variables también se pueden declarar junto con la definición de la estructura de la siguiente forma:

```
struct alumno{
      char nombre[30];
      char apellido[30];
      unsigned int legajo;
      float promedio;
      unsigned int ingreso
      unsigned int cantidad_materias;
}al1, Alumnos[100],*pal;
```

Acceso a los miembros

Para tener acceso a los miembros de la estructura se utilizan dos operadores: el operador de miembro de estructura ((.) también conocido como operador punto), y el operador de apuntador de estructura ((->) también conocido como el operador flecha).

El operador de miembro de estructura tiene acceso al miembro de la estructura usando el nombre de la estructura y el punto. Por ejemplo, para asignar el valor del miembro legajo en la estructura alumno al1 usando el operador punto, podemos hacer:

```
al1.legajo=46501;     //Operador miembro de estructura
```

De manera similar, el operador de apuntador de estructura tiene acceso al miembro de la estructura usando el nombre de un puntero y el operador flecha. Supongamos que el puntero pal ha sido inicializado apuntado a la estructura al1 mediante:

```
pal=&al1;
```

Podemos acceder al miembro promedio de `al1` usando este operador por medio de la siguiente expresión:

```
pal->promedio=8.5;  //Operador de apuntador de estructura
```

La expresión `pal->promedio` es equivalente a la expresión `(*pal).promedio` que desreferencia el puntero y tiene acceso al miembro promedio. Se necesita utilizar los paréntesis en este caso dado que el operador punto (.) tiene precedencia sobre el operador de desreferencia (*).

Inicialización

Las estructuras se pueden inicializar utilizando listas de inicialización de manera similar a los arreglos. Por ejemplo, la declaración:

```
struct alumno al1={"Eric","Cantona",46501,8.85,1995,32};
```

Crea la variable `al1` del tipo `struct  alumno` e inicializa el miembro nombre en “Eric”, el apellido en “Cantona”, el legajo en 46501, el promedio en 8.85, el ingreso en 1995 y cantidad_materias en 32. En caso de que la lista de inicialización contenga menos inicializadores que en la estructura, los miembros restantes se inicializaran en 0 o NULL dependiendo de su tipo.

Estructuras anidadas

Se denomina estructura anidada en el caso de que uno de los miembros que conforma una estructura es otra estructura. Considere las siguientes declaraciones de estructuras:

```
struct fecha{
      int dia;
      int mes;
      int anio;
};
struct tarjeta{
      long int numero;
      char tipo_cuenta;
```

```
        char nombre[80];
        float saldo;
        struct fecha ultimo_pago;
};
```

Como se puede observar en la declaración de la estructura tarjeta, el último de sus miembros “ultimo_pago” es del tipo previamente declarado “`struct fecha`“

A continuación, se presenta una posible inicialización para una variable (t1) del tipo struct tarjeta:

```
struct tarjeta t1={441199977,'A',"Naranja",11325.68,{22,10,2019}};
```

Para acceder a los miembros en estructuras anidadas se debe utilizar el operador punto tantas veces como sea necesario. Por ejemplo, para acceder a escribir el mes dentro del ultimo_pago en la estructura `t1` se puede hacer:

```
t1.ultimo_pago.mes=11;
```

Arreglos de estructuras

Usando la siguiente declaración junto con la definición de la estructura se puede generar un arreglo de 100 alumnos (100 estructuras del tipo `struct alumno`):

```
struct alumno{
        char nombre[30];
        char apellido[30];
        unsigned int legajo;
        float promedio;
        unsigned int ingreso
        unsigned int cantidad_materias;
} Alumnos[100];
```

Usando el índice del arreglo y el operador punto se puede acceder a los miembros de cada uno de los alumnos del listado.

```
//Asignamos el legajo y año de ingreso a cada alumno
for(i=0,j=60125;i<100;i++,j++)
{
       Alumnos[i].legajo=j;
       Alumnos[i].ingreso=2020;
}
```

Se pueden copiar estructuras entre sí de manera directa mediante sentencias del tipo:

```
Alumnos[10]=Alumnos[20];
```

Estructuras con funciones

Las estructuras se pueden pasar como parámetros a las funciones de manera directa: pasando como parámetro la propia estructura, por partes: pasando miembros de la estructura como parámetros a la función o bien de manera indirecta: por medio de un puntero a la estructura. En los dos primeros casos se están pasando los parámetros por valor, por lo cual la función estará trabajando sobre copias de los datos contenidos en la estructura original, sin capacidad de modificarla. En el último caso, cuando se pasa como parámetro un puntero a estructura, se implementa un pasaje de parámetros por referencia, pudiendo la función modificar los datos de la estructura original por conocer su ubicación en memoria.

Como ya se ha mencionado, los arreglos de cualquier tipo pasan como parámetros por referencia en las llamadas a función. Una buena alternativa para pasar arreglos a funciones por valor es incluirlos en una estructura y pasar la estructura como parámetro a la función, dado que las mismas pasan como parámetros por valor.

Se debe tener en cuenta que es más eficaz pasar estructuras en llamada por referencia donde se pasa tan solo la dirección de memoria, evitando copiar todos los miembros de la estructura como se hace en una llamada por valor.

A continuación se muestran ejemplos de pasaje de parámetros con estructuras por valor y por referencia:

Declaración de las funciones `fun1` y `fun2` que implementan llamadas por valor:

```
void fun1 (float prom);
void fun2 (struct alumno a1);
```

Y posibles llamadas a estas funciones:

```
fun1(al1.promedio);
fun2(al1);
```

Declaración de las funciones fun3 y fun4 que implementan llamadas por referencia:

```
void fun3 (float * prom);
void fun4 (struct alumno *p1);
```

Y posibles llamadas a estas funciones:

```
fun3(&al1.promedio);
fun4(&al1);
```

También pueden usarse estructuras como parámetro devuelto por las funciones, en cuyo caso las mismas deberían ser generadas dentro de la función y recibidas luego de la llamada:

La definición de la función sería del tipo:

```
struct alumno fun5()
{
      struct alumno alumno1;
      ....
      return alumno1;
};
```

Y la llamada a la función:

```
int main()
{
 struct alumno al1;
...
al1=fun5();
...
```

Campos de bits

Por medio del uso de los campos de bits, el lenguaje C ofrece al programador la posibilidad de optimizar el uso de la memoria especificando el número de bits en el cual se almacena un miembro int o unsigned en una estructura. Los miembros de los campos de bits deben ser declarados como int o unsigned.

Consideremos la definición de la siguiente estructura:

```
struct persona{
      unsigned edad:7;
      unsigned sexo:1;
      unsigned hijos:4;
};
```

La definición de la estructura contiene tres campos de bits: edad, sexo e hijos. Para declarar un campo de bits se utiliza un miembro unsigned o int con un signo de dos puntos (:) seguido de un número entero que representa el ancho del campo, es decir, la cantidad de bits en la que se almacena el miembro. Para el miembro edad se utilizan 7 bits, con lo cual el rango de representación es de 0 a 127 (2^7=128). De la misma forma, para el sexo se utiliza un bit, por lo cual puede almacenar solo dos valores 0 (M) o 1 (F). En el miembro hijos se utilizan 4 bits, siendo el rango de representación de 0 a 15 (2^4=16).

Uniones

Las uniones son un tipo de dato derivado, al igual que las estructuras, con la particularidad que todos los miembros que la conforman son almacenados en la misma posición de memoria. Los miembros que componen la union pueden ser de cualquier tipo y el espacio que se reserva

para alojarlos es el correspondiente al mayor de todos ellos. Esta particularidad de las uniones resulta útil para el ahorro de memoria, pero es responsabilidad del programador llevar la cuenta de con que miembro de la union se está operando en cada sección del código para evitar escribirlo de una forma y leerlo de otra.

Las uniones se declaran usando la palabra reservada `union` y respetando el mismo formato utilizado en las estructuras. A continuación, se puede observar el formato general para la declaración de una union:

```
union nombre{
      tipo elemento1;
      tipo elemento2;
      ……. . .
      tipo elementoN;
      };
```

En el código 7.1 se demuestra la importancia de acceder de manera correcta a los miembros de la union que comparten memoria.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  union numeros{
4        int num_entero;
5        float num_real;
6        };
7
8  int main()
9  {
10 union numeros num;
11 num.num_entero = 56;
12 printf("\n %d", num.num_entero);
13 printf("\n %5.2f", num.num_real);
14 num.num_real = 46.56;
15 printf("\n %5.2f", num.num_real);
16 printf("\n %d", num.num_entero);
17 return 0;
18 }
```

Código 7.1. *Ejemplo de uso de unión.*

En este ejemplo se declara una union de nombre `numeros` formada por un entero (`num_entero`) y un real (`num_real`). En la línea 11, por medio de la sentencia `num.num_entero = 56;`, se escribe en el miembro entero. En las líneas 12 y 13 se accede a ambos miembros de la union para leerlos e imprimirlos en pantalla. Cómo se puede observar en la ventana de salida de este código (Figura 7.1) solo se imprime de manera correcta el miembro entero que fue el último en el que se escribió. De la misma forma, las líneas 14, 15 y 16 presentan el mismo proceso, pero ahora accediendo de manera correcta al miembro real.

C:\Users\Uszu\Desktop\borrar\bin\Debug\borrar.exe

```
 56
 0.00
 46.56
 1111113073
Process returned 0 (0x0)   execution time : 3.452 s
Press any key to continue.
```

Figura 7.1. *Manejo de uniones.*

Ejemplo integrador

A continuación, se presenta un ejemplo integrador en el que se utilizan campos de bits y uniones, aprovechando las particularidades de ambos. Como se puede observar en la ventana de salida de ejecución (Figura 7.2), el código permite el ingreso repetido de caracteres, hasta ingresar la ‘x’, mostrando la representación binaria de cada caracter ingresado. Para lograr la manipulación a nivel de bit en el código, se implementa una union (`union charbits`) entre al caracter leído desde el teclado y un campo de bits que da acceso a cada uno de los bits de la representación (`struct byte`). Por medio de la función `decodifica` que recibe como parámetro un campo de bits, se realiza la impresión de los mismos.

La particularidad de las uniones en cuanto a compartir el espacio de memoria en el cual se ubican sus miembros resulta de gran utilidad en este ejemplo dado que nos permite guardar (escribir) el caracter leído desde teclado como un char y luego accederlo para lectura como un campo de bit.

```c
#include <stdio.h>
#include <conio.h>

struct byte {
unsigned int a:1;
unsigned int b:1;
unsigned int c:1;
unsigned int d:1;
unsigned int e:1;
unsigned int f:1;
unsigned int g:1;
unsigned int h:1;
};

```

```c
union charbits{
char ch;
struct byte bits;
}caracter;

void decodifica(struct byte b);

main(){

  puts("Ingrese caracteres. Para salir x");
  do{
          caracter.ch=getche();
          printf(":");
          decodifica(caracter.bits);
  }while(caracter.ch !='x');
}

void decodifica(struct byte b)
{

  printf("%2u%2u%2u%2u%2u%2u%2u%2u\n",b.h,b.g,b.f,b.e,b.d,b.c,b.b,b.a);
}
```

"C:\Users\Uszu\Dropbox\Programacion\Ejemplos clase\decodifica\bin\Debug\decodifica.exe"

```
Ingrese caracteres. Para salir x
a: 0 1 1 0 0 0 0 1
": 0 0 1 0 0 0 1 0
2: 0 0 1 1 0 0 1 0
@: 0 1 0 0 0 0 0 0
1: 0 0 1 1 0 0 0 1
a: 0 1 1 0 0 0 0 1
b: 0 1 1 0 0 0 1 0
c: 0 1 1 0 0 0 1 1
d: 0 1 1 0 0 1 0 0
e: 0 1 1 0 0 1 0 1
f: 0 1 1 0 0 1 1 0
A: 0 1 0 0 0 0 0 1
B: 0 1 0 0 0 0 1 0
C: 0 1 0 0 0 0 1 1
x: 0 1 1 1 1 0 0 0

Process returned 120 (0x78)   execution time : 42.241 s
Press any key to continue.
```

Figura 7.2. *Uniones y campos de bits*

Enumeraciones

La enumeración es un tipo de dato definido por el usuario formado por un conjunto de constantes enteras representadas por identificadores. Para generar las enumeraciones se utiliza la

palabra reservada `enum`. Estas constantes de enumeración son constantes simbólicas cuyos valores pueden ser definidos automáticamente. Estos valores se inician automáticamente desde cero y se van incrementando de a uno. Por ejemplo, la enumeración:

```
enum meses{ene,feb,mar,abr,may,jun,jul,ago,set,oct,nov,dic};
```

genera un nuevo tipo de datos `enum meses` en el cual los identificadores son definidos automáticamente con los enteros 0 a 11. Si queremos asignar los números 1 a 12 para los meses podemos hacer:

```
enum meses{ene=1,feb,mar,abr,may,jun,jul,ago,set,oct,nov,dic};
```

Los identificadores deben ser únicos y el valor de cada constante puede ser asignado explícitamente en la definición de la enumeración. Por otro lado, puede haber varios miembros de la enumeración que tengan el mismo valor.

A continuación, código 7.2, se presenta un ejemplo que utiliza la enumeración meses para imprimir un mensaje:

```
#include <stdio.h>
#include <stdlib.h>


enum meses{ene=1,feb,mar,abr,may,jun,jul,ago,set,oct,nov,dic};

int main()
{
enum meses mes;
for(mes=ene;mes<=dic;mes++)
  {
        if(mes==jul)
                    printf("Mes %d: Vacaciones de Invierno!!!",mes);
  }
return 0;
}
```

***Código 7.2.** Ejemplo de uso de enumeración*

Definición de tipos

La definición de tipos es un mecanismo por medio del cual podemos crear sinónimos para tipos de datos previamente definidos. Los nombres de los tipos de estructuras suelen definirse utilizando typedef para crear nombres de tipo más cortos. Por ejemplo, en la siguiente definición de tipo:

```
typedef int entero;
```

se define el nuevo nombre de tipo `entero` como un sinónimo para el tipo `int`. De ahora en adelante resulta indistinto el uso de `int o entero` para generar nuevas variables del tipo `int.`

De la misma forma, en la definición de tipo:

```
typedef struct tarjeta Tarjeta;
```

se define el nuevo nombre de tipo `Tarjeta` como un sinónimo para el tipo `struct tarjeta`. Es muy común encontrar typedef para definir un tipo de estructura de tal forma de no necesitar un rótulo de estructura. Por ejemplo, la definición:

```
typedef struct {
      long int num_tarjeta;
      char tipo_cuenta;
      char nombre [80];
      float saldo;
                } Tarjeta;
```

crea el tipo de estructura `Tarjeta`, sin necesidad de un enunciado por separado typedef. De ahora en adelante `Tarjeta` puede ser utilizado para crear variables de este tipo:

```
Tarjeta a;
```

Ejercicios

Estructuras compuestas

1) Se desea representar puntos sobre un plano de coordenadas reales `x` y, `y` asignarle a cada uno un carácter para identificarlos. Ejemplo: `int='a', x='23.5', y='5.14'`.
    a) Cree una estructura que permita representar estos puntos
    b) Declare tres variables del nuevo tipo creado, asígneles valores e imprima en pantalla los datos de cada punto.
    c) Cree una función que devuelva la distancia entre dos puntos
    d) Cree una función que tome un arreglo de puntos e imprima en pantalla cual es el más alejado del origen.

2) Realizar un programa que defina una tabla de proveedores, teniendo asignado cada proveedor un nombre, cantidad vendida del artículo, precio unitario (introducidos por teclado) e importe (calculado a partir de los datos anteriores). Se pretende visualizar los datos de cada proveedor, el importe total de compra, el nombre del proveedor más barato y el más caro.

3) Realizar un programa que defina una tabla de proveedores empleando una estructura que anida los datos del proveedor (nombre, dirección y teléfono), cantidad vendida, precio unitario e importe (calculado). Los datos no calculados se introducen por teclado. Se pretende visualizar en pantalla los datos de cada proveedor, el importe total de las compras y el nombre y teléfono del proveedor más barato.

4) Dada la siguiente definición:

```
struct datos
{
       int i;
       double f;
}Matriz[5][10];
struct datos *pd;
pd = Matriz;
```

a) Si el valor almacenado en `pd` es 2000, calcular en qué posición de memoria está el real `Matriz[3][6].f` . Considere que el tamaño de un `double` es de 8 bytes y el tamaño de un entero es de 4 bytes.
b) Realice un programa que muestre en pantalla la dirección en memoria de cada una de las estructuras que conforman la matriz.

5) Realizar un programa que permita visualizar en binario el código ASCII de los caracteres introducidos por teclado (hasta que un carácter sea el del cero). Se utilizará una unión que contenga un carácter y una estructura de campos de bits para contener un byte.

CAPÍTULO 8
Manejo de archivos

Marcelo A. Haberman

En este capítulo se explican las diferencias básicas entre los archivos de texto y binarios, así como el flujo de trabajo para su manipulación. Se presentan también las funciones de las bibliotecas de C más utilizadas para el manejo de los archivos, que facilitan su apertura, cierre, lectura, escritura y otras operaciones auxiliares.

Introducción

Los archivos son secuencias de bytes almacenadas en un sistema de archivos dentro de un dispositivo de almacenamiento masivo, como un disco duro magnético, un disco de estado sólido, una memoria SD, un DVD, etc. Esto nos permite resguardar información que durante la ejecución de un programa reside en la memoria principal de la computadora y que al finalizar la ejecución (o al apagar la computadora) se perdería.

Entre los diferentes sistemas de archivos y soportes físicos tenemos modos muy diferentes de almacenar los datos. Además, los archivos pueden almacenar información muy diferente entre sí (vídeo, audio, texto, hojas de cálculo, etc.). Pero, las funciones para el manejo de archivos que brinda la librería estándar de C (y el soporte que brinda el sistema operativo tras bambalinas) ayudan al programador a abstraerse de los detalles de implementación más tediosos y a pensar al archivo como una secuencia de bytes o caracteres.

Los archivos en general se identifican por un nombre y una ruta que debe seguir las reglas del sistema de archivos y/o del sistema operativo. Por ejemplo, en un sistema tipo Unix como Linux o macOS una ruta completa para un archivo puede ser:

/Materias/Programación/**archivo.txt**

Mientras que en Windows las rutas tienen otra apariencia:

C:\Materias\Programación**archivo.txt**

En ambos casos el nombre del archivo sería **archivo.txt** y las rutas indican que está almacenado en una carpeta llamada “Programación”, la cual está dentro de otra carpeta “Materias”. En la ruta tipo Unix esta carpeta está en la raíz del sistema de archivos y en la ruta tipo Windows está en la raíz de la unidad o disco “C:”. Dependiendo para qué sistema

estemos programando tenemos que considerar esos detalles al manipular los archivos desde nuestro programa.

Sobre un archivo podemos realizar principalmente dos operaciones:

- Leer datos guardados en este y almacenarlos en la memoria principal para poder procesarlos con el programa.
- Escribir en el archivo los datos almacenados en la memoria principal que desean ser resguardados.

En este capítulo veremos las distintas maneras en que pueden realizarse las operaciones de lectura y escritura, distinguiendo entre las operaciones utilizadas con archivos de texto y con archivos binarios.

Archivos de texto vs. archivos binarios

Al trabajar con archivos haremos una gran distinción entre dos tipos distintos: los archivos de texto y los archivos binarios. En la primera categoría vemos al archivo como una secuencia de caracteres, que forman líneas terminadas en un especificador de *fin de línea*[16]. La ventaja de estos archivos es que pueden abrirse con cualquier editor de texto plano (como el bloc de notas de Windows) y cualquier información almacenada es legible dado que fue almacenada como texto. Además, son medianamente portables si se conoce el sistema de codificación utilizado (ASCII/ANSI, UTF8, UTF16, etc.) y el tipo de *fin de línea*. La desventaja de este tipo de archivos es que suelen ser ineficientes en cuanto al uso del espacio de almacenamiento, sobre todo al almacenar números de muchas cifras significativas y, además, si la información es preferentemente numérica hay que hacer una conversión de número a texto al escribir y de texto a número al leer del archivo. En la figura 8.1 puede verse un archivo de texto abierto con el bloc de notas, en el cual se pueden distinguir cuatro líneas, cada una con un valor numérico. A la derecha se observa el contenido del archivo a nivel de bytes (en hexa), donde se ven los caracteres numéricos (0x30 al 0x39) y, algo propio del sistema de archivos de windows, los fines de línea codificados como dos caracteres: el retorno de carro `'\r'` (0x0D) y el salto de línea `'\n'` (0x0A).

[16] Este especificador depende del sistema. En Unix coincide con el caracter `'\n'` (0x0A) mientras que en Windows cada *fin de línea* está compuesto por dos caracteres: `'\r'` (0x0D) y `'\n'` (0x0A). Afortunadamente los programadores en general no deben preocuparse por esto ya que las funciones de entrada/salida (E/S) para archivos de texto se encargan de convertir entre el *fin de línea* en el archivo y el caracter de nueva línea `'\n'` en memoria.

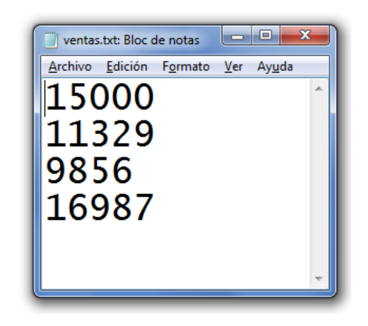


Figura 8.1: *Vista de un archivo de texto en editor de texto plano y contenido en bytes del archivo.*

Los archivos binarios, por el contrario, almacenan una secuencia de bytes imitando la representación y organización de los bytes en memoria principal. Esto hace que el archivo no sea fácilmente legible mediante editores de texto, sino que para interpretar un archivo binario deberá ser procesado por un programa que sepa cómo está codificada la información. El punto a favor de los archivos binarios es (en general) un menor tamaño y la escritura/lectura directa sin tener que convertir o formatear los datos hacia/desde texto. En la figura 8.2 puede observarse un archivo binario abierto con el bloc de notas que guarda los mismos números que el archivo de la figura 8.1. La información mostrada como texto es inútil, ya que los bytes corresponden a la representación en memoria de los cuatro valores en variables `unsigned short` (binario puro de 16 bits - little endian), es decir que cada par de bytes representa uno de los cuatro valores.

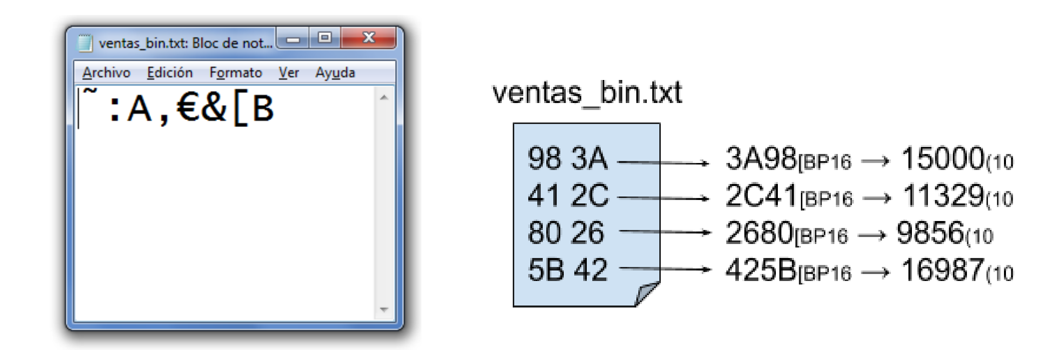


Figura 8.2: *Vista de un archivo binario en editor de texto plano y contenido en bytes del archivo. Nótese que la extensión ".txt" no implica que el archivo sea de texto, sino que sirve para que el sistema operativo abra el archivo con la aplicación asociada a tal extensión.*

Flujo de trabajo con archivos

El flujo de trabajo con archivos en C sigue una secuencia específica:

1. Definir una variable *manejador de archivo*
2. Abrir el archivo
3. Realizar operaciones sobre el archivo de lectura y/o escritura, y otras operaciones secundarias.
4. Cerrar el archivo

Manejador de archivo

Las variables manejadoras de archivos permiten identificar en el código a los distintos archivos y representan el mecanismo para que los programadores puedan pasar referencias de los mismos a las funciones que acceden y manipulan el contenido de los archivos.

En el archivo de cabecera `stdio.h` se define el tipo `FILE`, el cual es una estructura que permite llevar el control de un archivo, pero como programadores no manipularemos directamente ninguna variable de este tipo, sino un puntero que referencia a una de estas estructuras, que permitirá intercambiar y modificar información del archivo con las distintas funciones.

Resumiendo, lo que necesitamos en primera instancia es definir una variable manejadora de archivo de tipo `FILE*` (puntero a `FILE`), la cual apuntará a una estructura en memoria que se creará cuando abramos el archivo.

```
FILE* manejador_archivo;
```

Apertura del archivo

La acción de abrir un archivo se realiza mediante una invocación a la función `fopen()` cuyo prototipo es el siguiente:

```
FILE *fopen (char * nombre_completo_archivo, char * modo);
```

El primer parámetro es una cadena de caracteres con la ruta completa del archivo que deseamos abrir. Si solo se indica el nombre del archivo, el sistema operativo lo ubicará en alguna carpeta particular como la carpeta donde reside el archivo ejecutable o el proyecto con el código fuente.

El segundo parámetro es una cadena donde se indica el modo de apertura, que determinará el alcance de las acciones que podemos realizar sobre el archivo:

Modo de Apertura	Significado
`"a"`	Abre el archivo para añadir datos al final del archivo. Si el archivo no existe, se crea.
`"r"`	Abre el archivo para lectura. El archivo debe existir.
`"w"`	Abre el archivo para escritura. Los datos se escriben desde el principio. Si el archivo no existe, se crea. Si existe se elimina su contenido.
`"a+"`	Abre el archivo para lectura y para añadir datos al final del mismo. Si el archivo no existe, se crea.
`"r+"`	Abre el archivo para lectura y escritura. Los datos se escriben desde el principio sobreescribiendo los datos preexistentes. El archivo debe existir.
`"w+"`	Abre el archivo para lectura y escritura. Los datos se escriben desde el principio y si el archivo no existe se crea. Si existe se elimina su contenido.

Todas las cadenas de la tabla provocan la apertura de un archivo de texto. Para abrir un archivo binario, se debe agregar la letra `b` a la cadena con el modo deseado.

La función intenta reservar los recursos necesarios para trabajar con el archivo, entre otras cosas crea la estructura `FILE` necesaria para manipularlo y retorna un puntero a la misma si no ocurre ningún error en dicho proceso. En caso de producirse algún error la función retorna la constante `NULL` (puntero nulo), por lo que es una buena práctica chequear que el valor retornado sea distinto de `NULL` antes de realizar cualquier procesamiento sobre el archivo.

Por ejemplo, para abrir **ventas.txt** como archivo de texto en modo actualización (añadir datos al final) podría usarse el siguiente código:

```
manejador_archivo = fopen("ventas.txt", "a");
```

Y luego para chequear errores en la apertura

```
if(manejador_archivo == NULL){
    puts("Error al abrir...");
    ...
}
```

También suelen hacerse ambas cosas a la vez, teniendo en cuenta que la operación de asignación da como resultado el valor asignado

```
if((manejador_archivo = fopen("ventas.txt", "a")) == NULL){
    puts("Error al abrir...");
    ...
}
```

Cierre del archivo

En el momento en que se decide que no se trabajará más con un archivo conviene cerrarlo. Esta acción implica terminar de transferir todos los datos que temporalmente permanezcan en la memoria principal al medio físico, finalizando prolijamente el trabajo sobre el archivo y liberando los recursos utilizados. Este procedimiento se realiza mediante una invocación a la función `fclose()` cuyo prototipo es el siguiente:

```
int fclose(FILE * manejador);
```

La función recibe el manejador del archivo (previamente abierto con `fopen()`) y si logra cerrarlo con éxito devuelve 0.

Por ejemplo, se puede cerrar el archivo abierto en la sección anterior de la siguiente manera

```
fclose(manejador_archivo);
```

Procesamiento secuencial

Veremos que hay numerosas funciones para leer/escribir en archivos. Todas estas funciones, aunque muy distintas entre sí, comparten una misma filosofía: además de realizar su tarea específica de leer o escribir lo que corresponde en cada caso, también actualizan automáticamente

la posición o cursor dentro del archivo. Esto quiere decir que como programadores podemos invocar sucesivamente a estas funciones abstrayéndonos de la posición dentro del archivo, ya que la misma es actualizada de manera automática.

Por ejemplo, en la figura 8.3 vemos el pseudocódigo de un programa que realiza varias escrituras sucesivas a un archivo y cómo se va modificando el contenido del mismo, a la vez que se actualiza la posición del cursor de escritura luego de cada llamada. En la figura 8.4 se ilustra la situación homóloga para lecturas sucesivas.

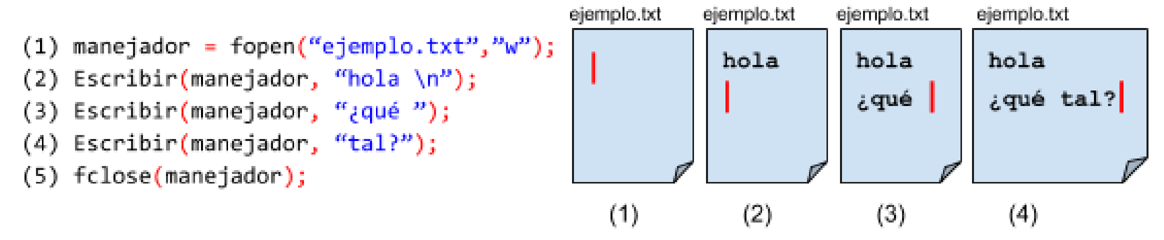


***Figura 8.3:** Pseudocódigo y estado del archivo luego de la ejecución de cada línea. La línea roja indica la posición del cursor.*

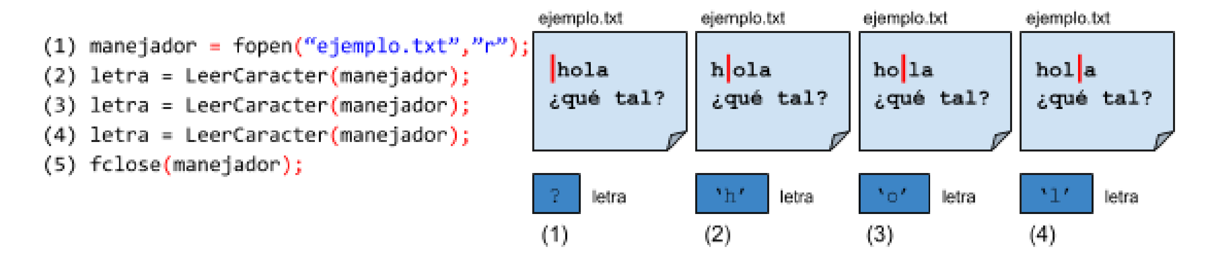


***Figura 8.4:** Pseudocódigo y estado del archivo y variables luego de la ejecución de cada línea. La línea roja indica la posición del cursor.*

Funciones de lectura/escritura para archivos de texto

Veremos que hay tres maneras de manipular los archivos de texto: escribiendo o leyendo de a un caracter por vez, de a una cadena/línea por vez o con formatos personalizados por el programador. Todas las funciones recibirán como parámetro el manejador del archivo y luego de realizar la operación correspondiente actualizarán la posición o cursor luego de los datos leídos o escritos.

Lectura de caracteres

Para leer un caracter de un archivo se utiliza la función `fgetc()` o su alias `getc()`, cuyo prototipo es:

```
int fgetc(FILE * manejador);
```

La función lee el caracter en la posición actual, avanzando luego la posición al siguiente caracter y retornando el caracter leído como un `unsigned char` convertido a `int`. En caso de llegar al final del archivo retornará la constante `EOF`.

El código 8.1 es un ejemplo de cómo puede leerse un archivo completo y mostrarlo en la consola con la función `fgetc()`. La clave de todo está en lo que ocurre en la línea 13: al evaluarse la condición del `while` se ejecuta la expresión `c=fgetc(manejador_archivo)` que intentará leer un carácter y almacenarlo en la variable `c`. El valor asignado en `c` se compara luego con `EOF` para chequear si se terminó de leer el archivo, de ser así se interrumpe el bucle, se cierra el archivo y el programa termina. Si no se llega al final, el carácter leído se muestra en pantalla continuando con el bucle de lectura. Puede parecer extraño que la variable `c` sea de tipo `int`, pero esto se hace para poder representar cualquier carácter leído retornado dentro del rango entre 0 y 255, y a su vez poder recibir el valor de `EOF` (-1) como un valor fuera del rango de caracteres válidos.

```
1       #include <stdio.h>
2
3       int main()
4       {
5               FILE* manejador_archivo;
6               int c;
7               if((manejador_archivo=fopen("ventas.txt","r")) == NULL){
8                       printf("ERROR");
9                       return 1;
10              }
11
12              while((c=fgetc(manejador_archivo)) != EOF)
13                      printf("%c", c);
14
15              fclose(manejador_archivo);
16              return 0;
17      }
```

Código 8.1: *Ejemplo de lectura de un archivo de texto, carácter por carácter, e impresión en pantalla.*

Escritura de caracteres

Para escribir un caracter en un archivo se utiliza la función `fputc()` o su alias `putc()`, cuyo prototipo es:

```
int fputc(int car, FILE * manejador);
```

Esta función intenta escribir el caracter `car` (convirtiendo primero a `unsigned char`) en el archivo que referencia `manejador`. Si lo logra retorna el mismo caracter, si falla retorna `EOF`.

El código 8.2 crea el archivo **mitexto.txt** si no existe, o lo trunca si existe, y guardará en el los caracteres ingresados por el usuario, hasta que se ingrese un salto de línea (se presiona la tecla *enter*) que también será escrito en el archivo.

```c
#include <stdio.h>

int main()
{
    FILE* f;
    int c;
    if((f=fopen("miarchivo.txt","w")) == NULL){
           printf("ERROR");
           return 1;
    }

    do {
           c = getchar();
           putc(c, f);
    } while (c != '\n');

    fclose(f);
    return 0;
}
```

Código 8.2: *Ejemplo de escritura de un archivo de texto con la información ingresada en tiempo de ejecución por el usuario, carácter por carácter.*

Lectura de cadenas

Para leer una línea de un archivo de texto se utiliza la función `fgets()` , cuyo prototipo es:

`char* fgets(char* cadena, int tam, FILE * manejador);`

La función lee una serie de caracteres que irá almacenando en el arreglo `cadena`, la lectura se interrumpe cuando se cumple alguna de las siguientes condiciones:

- se lee un *fin de línea*, que en la cadena se guarda como un carácter `‘\n’`.
- se leen `tam`-1 caracteres (en este caso no se guarda un `‘\n’` en la cadena)
- se llega al final del archivo (tampoco se guarda un `‘\n’` en la cadena)

En los tres casos se guarda en el arreglo el carácter `‘\0’`, luego de los caracteres leídos, para finalizar correctamente la cadena de caracteres. Es importante notar que en el archivo no se almacenan caracteres de finalización de cadena.

El valor de retorno de esta función será un puntero igual a `cadena` en caso de lectura exitosa. En caso de llegar al final del archivo y que ningún carácter haya podido ser leído, la función devuelve `NULL`.

El código 8.3 abre el archivo **prueba.txt** y lo va leyendo e imprimiendo en pantalla línea por línea hasta que se llega al final del mismo y ya no haya más que leer (la llamada a `fgets()` de la línea 16 retorna `NULL`).

```
1	#include <stdio.h>
2	#define TAM 80
3	int main()
4	{
5		FILE* f;
6		char s[TAM], *ret;
7
8		if((f=fopen("prueba.txt","r")) == NULL){
9			printf("ERROR");
10			return 1;
11		}
12
13		ret=fgets(s,TAM,f);
14		while (ret != NULL){
15			printf(s);
16			ret=fgets(s,TAM,f);
17		}
18
19		fclose(f);
20		return 0;
21	}
```

Código 8.3: *Ejemplo de lectura de un archivo de texto, línea por línea. La información leída se imprime en pantalla.*

Escritura de cadenas

Para escribir una cadena de caracteres en un archivo de texto se utiliza la función `fputs()` , cuyo prototipo es:

```
int fputs(const char* cadena, FILE * manejador);
```

La función escribe en el archivo los caracteres almacenados en `cadena`, sin incluir el carácter de final de cadena `‘\0’`.

El valor de retorno es un entero no negativo en caso de éxito o `EOF` en caso de error.

El código 8.4 crea o trunca el archivo **prueba.txt** y escribe las líneas ingresadas por el usuario hasta que el mismo ingresa la cadena `“salir”`. En la línea 17 se hace un llamado adicional a `fputs()` para escribir en el archivo un fin de línea luego de la cadena escrita en la línea 16.

```
#include <stdio.h>
#include <string.h>
#define TAM 80
int main()
{
        FILE* f;
        char s[TAM],*ret;

        if((f=fopen("prueba.txt","w")) == NULL){
                printf("ERROR");
                return 1;
        }

        scanf("%79[^\n]",s);fflush(stdin);
        while (strcmp(s,"salir") != 0){
                fputs(s,f);
                fputs("\n",f);
                scanf("%79[^\n]",s);fflush(stdin);
        }

        fclose(f);
        return 0;
}
```

Código 8.4: *Ejemplo de escritura de un archivo de texto con la información ingresada en tiempo de ejecución por el usuario, línea por línea.*

Lectura con formato

La lectura con formato permite extraer e interpretar, de entre el texto del archivo, información de distintos tipos y almacenarla en las variables correspondiente. La función que se utiliza para la lectura con formato es `fscanf()` que funciona igual al `scanf()` pero en lugar de leer de la entrada estándar (el teclado) lee de un archivo cuyo manejador se pasa como primer parámetro. Su prototipo es el siguiente:

```
int fscanf(FILE * manejador,const char* cadena_formato, ... );
```

El valor de retorno es un entero que devuelve la cantidad de valores leídos (puede que se lean menos que los deseados por errores en el formato). En caso de alcanzar el final del archivo y no haber podido leer ningún valor retorna `EOF`.

El código 8.5 muestra un ejemplo de uso de `fscanf()` para leer enteros sin signo del archivo de texto **ventas.txt**, el cual tiene almacenado un entero por línea como se ve en la figura 8.1. El código realiza una sumatoria de los números almacenados en el archivo e imprime el resultado en pantalla.

```
#include <stdio.h>
int main()
{
        FILE* f;
        int ret;
        unsigned item, suma=0;

        if((f=fopen("ventas.txt","r")) == NULL){
                printf("ERROR");
                return 1;
        }

        ret = fscanf(f,"%u\n",&item);
        while (ret != EOF){
                suma += item;
                ret = fscanf(f,"%u\n",&item);
        }
        printf("Total = %u",suma);
        fclose(f);
        return 0;
}
```

Código 8.5: *Ejemplo de lectura de un archivo de texto considerando el formato, un valor entero sin signo por línea. Los valores leídos son sumados y el resultado es mostrado al usuario.*

Escritura con formato

La escritura con formato permite almacenar en el archivo de texto información de distintos tipos con el formato deseado. La función que se utiliza para la lectura con formato es `fprintf()` que funciona igual al `printf()` pero en lugar de escribir en la salida estándar (la pantalla) lo hace en un archivo de texto cuyo manejador se pasa como primer parámetro. Su prototipo es el siguiente:

```
int fprintf(FILE * manejador,const char* cadena_formato, ... );
```

El valor de retorno es un entero con la cantidad de caracteres escritos o un valor negativo en caso de error.

El código 8.6 muestra un ejemplo de uso de `fprintf()` para guardar los enteros de un arreglo en el archivo de texto **ventas.txt**, generando un archivo como el de la figura 8.1.

```
#include <stdio.h>
int main()
{
        unsigned short ventas[]={15000, 11329, 9856, 16987};
        FILE* f;
        int i;

```

```
        if((f=fopen("ventas.txt","w")) == NULL){
                printf("ERROR");
                return 1;
        }

        for(i=0;i<4;i++){
                fprintf(f,"%u\n",ventas[i]);
        }
        fclose(f);
}
```

Código 8.6: *Ejemplo de escritura de un archivo de texto con formato: un entero sin signo por línea.*

Funciones de lectura/escritura para archivos binarios

En la lectura y escritura de archivos binarios se realizan copias exactas de bloques de bytes entre estos archivos y la memoria principal. Son ideales para resguardar grandes volúmenes de información sin preocuparse por cuestiones de formato y portabilidad. Estas funciones, al igual que las que se vieron anteriormente actualizan automáticamente la posición del cursor luego de los datos leídos o escritos.

Lectura de archivos binarios

La lectura de información de un archivo binario se realiza mediante la función `fread()`, cuyo prototipo es el siguiente;

```
size_t fread(void* memoria,size_t tam_bloq, size_t num_bloq, FILE *
manejador);
```

El primer parámetro es un puntero con la dirección del byte donde comienza la zona de memoria en la cual se guardarán los datos leídos, notar que es de tipo `void*`, que es un puntero genérico que permite aceptar cualquier tipo de puntero que se pase como parámetro. El último parámetro es el manejador del archivo.

El segundo y el tercer parámetro, al igual que el valor de retorno, son de tipo `size_t`, que es un alias de algún tipo entero sin signo definido por la implementación, típicamente `unsigned int`, que generalmente se utiliza para indicar el tamaño de objetos de un programa.

El parámetro `tam_bloq` es entonces un entero que indica el tamaño en bytes de cada bloque de información a leer (usualmente es el tamaño, devuelto por `sizeof`, de una estructura o tipo de dato básico), mientras que el tercer parámetro `num_bloq` es otro entero que indica cuántos bloques de datos se intentarán leer.

El valor de retorno indica cuántos bloques fueron correctamente leídos del archivo y guardados en memoria a partir de la dirección pasada en el primer parámetro. Este puede ser igual a

`num_bloq` en caso de haber leído todos los bloques deseados o menor en caso de que finalice el archivo u ocurra un error.

En el código 8.7 se muestra un ejemplo de cómo leer los datos de un archivo binario, que almacena enteros cortos sin signo (`unsigned short`) y guardar esos datos en memoria. En la línea 5 se define el arreglo `ventas` con capacidad para 100 `unsigned short`, donde se guardarán los valores leídos del archivo binario **ventas_bin.txt** (abierto en la línea 9). La lectura de todo el archivo al arreglo se realiza en la línea 14, con una única llamada a `fread()`, donde se intentan leer 100 valores `unsigned short` (el máximo que se puede almacenar en el arreglo). El valor de retorno que se almacena en la variable `n`, indicará cuántos `unsigned short` se leyeron del archivo y se guardaron en el arreglo `ventas`. Luego de la lectura se indica al usuario cuantos valores fueron leídos y se muestra la lista.

```
1	#include <stdio.h>
2	#define TAM 100
3	int main()
4	{
5		unsigned short ventas[TAM]={};
6		FILE* f;
7		unsigned n,i;
8	
9		if((f=fopen("ventas_bin.txt","rb")) ==NULL){
10			printf("ERROR");
11			return 1;
12		}
13	
14		n = fread(ventas,sizeof(unsigned short),TAM,f);
15	
16		printf("Se leyeron %d enteros:\n", n);
17		for(i=0;i<n;i++)
18			printf("%d\n",ventas[i]);
19	
20		fclose(f);
21		return(0);
22	}
```

Código 8.7: *Ejemplo de lectura de un archivo binario que almacena enteros tipo* `unsigned short`.

El resultado de correr el código 8.7 con el archivo de la figura 8.2 se muestra en la figura 8.5.

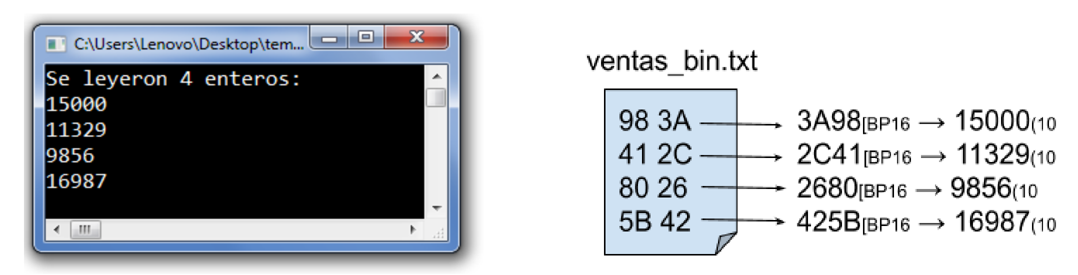


Figura 8.5: *Resultado de correr el programa de ejemplo con el archivo de la fig. 9.2.*

Escritura en archivos binarios

La escritura de información en un archivo binario se realiza mediante la función `fwrite()`, cuyo prototipo es el siguiente;

```
size_t fwrite(void* memoria,size_t tam_bloq, size_t num_bloq, FILE* manejador);
```

El primer parámetro es un puntero con la dirección del byte donde comienza la zona de memoria cuyos bytes se desean copiar al archivo, cuyo manejador se pasa en el último parámetro.

El segundo parámetro `tam_bloq` indica el tamaño en bytes de cada bloque de información a escribir (usualmente es el tamaño de una estructura o tipo de dato básico), mientras que el tercer parámetro `num_bloq` indica cuántos bloques de datos se intentarán escribir.

El valor de retorno indica cuántos bloques fueron correctamente copiados al archivo.

El código 8.8 muestra un ejemplo de cómo puede copiarse todo un arreglo desde la memoria a un archivo binario por medio de `fwrite()`. En este caso se generará el archivo de la figura 8.2 usado en ejemplos previos.

```
#include <stdio.h>
#define TAM 4
int main()
{
        unsigned short ventas[TAM]={15000,11329,9856,16987};
        FILE* f;
        unsigned n,i;

        if((f=fopen("ventas_bin.txt","wb")) ==NULL){
                printf("ERROR");
                return 1;
        }

        n = fwrite(ventas,sizeof(unsigned short),TAM,f);

        printf("Se escribieron %d enteros.", n);

        fclose(f);
        return(0);
}
```

Código 8.8: *Ejemplo de escritura de un archivo binario que almacenará enteros tipo `unsigned short`.*

Otras operaciones con archivos

Reposicionamiento del cursor (acceso directo)

El procesamiento secuencial explicado al comienzo del capítulo no es la única manera en la que podemos manipular un archivo. También es posible posicionar el cursor en cualquier lugar que se desee del archivo y leer/escribir en dicha posición sin tener que hacerlo secuencialmente desde el principio. Esto se realiza con la función `fseek()`.

```
size_t fseek(FILE *archivo, long posicion, int origen);
```

Donde el parámetro `origen` específica donde comienza la búsqueda tomando el valor de una de tres constantes posibles:

– `SEEK_CUR`: Posición actual del cursor..

– `SEEK_END`: Final del archivo.

– `SEEK_SET`: Comienzo del archivo

El parámetro `posición` determina un desplazamiento en bytes respecto de `origen`, siendo los valores positivos desde `origen` hasta el final, y los negativos hacia el comienzo del archivo.

La función retorna 0 si el cursor del archivo se desplazó satisfactoriamente y un valor distinto de 0 si fracasó.

Veamos algunos ejemplos suponiendo un archivo binario que tiene almacenado un arreglo de tipo `struct producto`:

```
fseek(fp,0,SEEK_SET); /* Va al comienzo del archivo */
fseek(fp,10*sizeof(struct producto),SEEK_SET);  /* Va al 11º producto */
fseek(fp,2*sizeof(struct producto),SEEK_CUR);   /* Salta dos productos */
fseek(fp,-sizeof(struct producto),SEEK_END);    /* Se posiciona para leer
                                                   el último producto */
```

Obtención de la posición del cursor

La posición del cursor en el archivo (medida en bytes desde el comienzo del mismo) se puede obtener con la función `ftell()`

```
int ftell (FILE *archivo);
```

La función retornará los bytes que hay desde el comienzo del archivo, o -1 en caso de error.

Verificación de llegada al final del archivo

Determinar si se alcanzó la posición final del archivo tiene particular interés en la lectura de archivos tanto de texto como binarios, permitiendo implementar bucles del tipo:

```
Mientras( no llego al final del archivo)
    Seguir leyendo.
```

Por este motivo las funciones de lectura retornan las constantes `EOF` (`fgetc` y `fscanf`) o `NULL` (`fgets`) en caso de intentar leer en la posición final.

En el caso de la lectura binaria es posible deducir que se llegó al final del archivo por el valor de retorno de `fread`, que será menor al número de bloques que se deseaban leer.

Adicionalmente a estos métodos contamos con la función `feof()`:

```
int feof(FILE *archivo);
```

que retornará un valor distinto de cero si se alcanzó el final del archivo o cero en caso contrario.

Ejercicios

1) a) Escriba un programa que lea sucesivas líneas de texto ingresadas por el usuario y las almacene en un archivo de texto. El programa termina cuando el usuario ingresa una línea vacía.
   b) Escriba un segundo programa que cuente el número de líneas presentes en un archivo de texto cuyo nombre es ingresado por teclado. Utilice el archivo generado en el inciso anterior.

2) Crear un programa que copie el contenido de un archivo en otro,
   a) Utilizando las funciones de E/S de caracteres `fgetc()` y `fputc()`.
   b) Utilizando las funciones de E/S de cadenas de caracteres `fgets()` y `fputs()`.
   c) Utilizando las funciones de E/S formateadas `fscanf()` y `fprintf()`.

3) a) ¿Qué tamaño tendrá un archivo de texto donde se escriba el número real 163322,2274 y qué tamaño tendría un archivo binario que almacene el mismo número?
   b) Enumere las funciones útiles para leer y almacenar datos en un archivo binario y explique sus parámetros.

4) a) Realizar un programa que defina una tabla de proveedores empleando una estructura que anida los datos personales del proveedor (nombre, dirección y teléfono), cantidad vendida, precio unitario e importe (calculado). Los datos no calculados se introducen por teclado. Guarde el arreglo de Proveedores en un archivo binario.
   b) Cree un nuevo programa que permita cargar los valores del archivo binario creado en el inciso anterior en un arreglo de Proveedores y los imprima en pantalla.

5) Utilice la función `ftell()` para analizar el archivo creado en el problema 4.a y cree un programa que imprima en pantalla el tamaño en bytes total del archivo, cuántas estructuras

Proveedor contiene, y luego que imprima la primera estructura, la última, y la que se encuentra a la mitad, recuperándolas directamente desde el archivo (es decir, sin pasar todos los datos a un arreglo).

6) Escriba un programa que contabilice el número de veces que una palabra se encuentra en un archivo de texto, aún cuando ésta forme parte de otra palabra.

Ejercicios integradores

7) Realizar un programa que dado un archivo de texto origen, genere un nuevo archivo de salida que contenga el mismo texto que el de origen, pero con sus líneas de texto, alineadas a izquierda, derecha o centro, según lo indique el usuario por línea de comandos con el carácter `'i'`, `'d'`, o `'c'`. Considere el largo máximo de una línea de caracteres igual a 80. Los nombres de los archivos origen y destino deben ingresar también por línea de comandos.

Ejemplo de línea de comandos:
>>prueba origen.txt destino.txt i

8) Según la fórmula de Taylor, podemos expresar la función exponencial e^x mediante la siguiente serie:

$$e^x = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \ldots$$

Realice un programa que:
En forma reiterada pida al usuario el ingreso del valor del exponente (x), o Salir del programa.
Calcule la función exponencial con la máxima cantidad de cifras significativas mediante la serie de Taylor. Note que para lograrlo puede sumar términos hasta que el número no cambie.
Calcule la función exponencial utilizando la función `exp ()` incluída en `math.h` .
Genere un archivo de texto de nombre *exponencial.txt* y guarde en cuatro columnas: el valor del exponente, el valor obtenido en el punto b, el valor obtenido en c y el error relativo.
La opción Salir mostrará en pantalla el contenido del archivo generado antes de terminar el programa.
NOTA: Tenga presente en la resolución del algoritmo que:

$$\frac{x^i}{i!} = \frac{x^{i-1}}{i-1!} * \frac{x}{i}$$

9) Realizar un programa que ordene alfabéticamente las líneas de un archivo origen y las guarde ordenadas en un archivo destino. Los nombres de ambos archivos deben pasarse

como argumentos de `main()`, por línea de comandos del sistema operativo. Utilice una matriz de caracteres y aplique un algoritmo de ordenamiento. Se recomienda el uso de las funciones `fgets()` y `fputs ()`.
Recuerde que, en el algoritmo de ordenamiento, deberá usar una función como `strcoll()` para para comparar alfabéticamente las cadenas, y la función `strcpy()` para reemplazar su posición al ordenarlas.

CAPÍTULO 9
Memoria dinámica

Federico N. Guerrero

Tamaños definidos en tiempo de ejecución

Hasta ahora, todos los tipos de datos (estándares o definidos por el usuario) y arreglos que podían ser declarados, debían ser declarados con un tamaño fijo en tiempo de compilación[17]. Es decir que el tamaño de todos los datos contenidos en un programa debería ser siempre conocido antes de su ejecución.

Esto hace muy seguro al leguaje C en cuanto al uso de memoria, pero no siempre resulta en la forma óptima o más práctica de utilizar el espacio. Por ejemplo, una de las primeras tareas que enfrentamos al comenzar a programar es leer el texto que un usuario ingresa, como su nombre. En ese contexto nos preguntamos cuántos caracteres reservar para el nombre, que coincidirá con la cantidad de letras más uno para el carácter nulo de la cadena. Sin embargo, no podemos conocer esto en tiempo de compilación. De hecho, podemos asegurar que será siempre distinto para distintas personas. La estrategia que adoptamos es suponer que habrá siempre una cota máxima y todos los nombres tendrán menos letras. Esto funciona hasta que aparece el Sr. Hubert Blaine Wolfeschlegelsteinhausenbergerdorff y arruina nuestros planes.

Algunos ejemplos más pertinentes son el ingreso de grandes volúmenes de datos por algún periférico; el manejo de archivos guardados en almacenamiento permanente, que pueden tener tamaños muy grandes y en general son desconocidos por el programa hasta que comienza a manipularlos; los programas numéricos que utilizan matrices o arreglos, en ocasiones cuyos tamaños dependen de resultados de cálculos intermedios; programas que deben adaptarse eficientemente a algunos parámetros de configuración. En todas estas ocasiones, y muchas más, puede convenir declarar el tamaño de las variables en tiempo de ejecución.

[17] Suele distinguirse el “tiempo de compilación” al referirse al momento cuando el código se compila y toda la información disponible proviene del propio código, del “tiempo de ejecución” cuando el programa ya está compilado y corriendo en una computadora, cuando información adicional que se desconocía parcialmente al momento de escribir el código puede provenir de una entrada.

Un camino “poco recomendable”

La primera estrategia para resolver este tipo de problema sería utilizar un elemento relativamente “nuevo” del lenguaje C, que es el *arreglo de tamaño variable* o VLA por sus siglas en inglés. Los VLAs fueron introducidos en el estándar C99. El lenguaje C es regulado por una serie de estándares con los que deben cumplir todos los compiladores (cumplir con ese estándar es lo que hace al compilador un compilador de C). Estos estándares evolucionan con el tiempo y en el año 1999 se definió esta versión que, entre otros agregados, posibilitaba definir un arreglo como en el código 9.1.

```
int n;
printf("Ingrese la cantidad de elementos del arreglo: ");
scanf("%d",&n);
int arr[n];
```

Código 9.1. *Declaración de un arreglo de largo variable.*

Es decir, que puede declararse el arreglo en base a una variable cuyo valor puede ser asignado en tiempo de ejecución, y desconocerse completamente en tiempo de compilación.

Esta herramienta es muy sencilla y por supuesto puede aplicarse en muchas ocasiones, pero es desaconsejable como “única herramienta” por varios motivos de menor o mayor peso.

El primer motivo es que más adelante, en el estándar C11 (año 2011), se estableció que esta característica sería opcional. Los compiladores ya no tienen la obligación de implementarla y por lo tanto el código que escribamos usando VLAs podría no servir en otro compilador. El segundo motivo, más significativo, es que al intentar reservar espacio de esta manera no hay forma de evaluar si efectivamente existe la cantidad de espacio necesaria. Si no lo hay, el programa directamente falla. Otras formas de reservar memoria, que veremos en las secciones siguientes, permiten conocer el resultado de la operación de reserva y elegir el camino de ejecución más apropiado si la memoria no está disponible.

Uso básico de memoria dinámica

La memoria dinámica se utiliza a través de una serie de funciones de la librería estándar. Antes de entrar en detalles sobre las funciones en sí, recordemos que el objetivo es obtener un espacio de memoria para almacenar datos durante la ejecución del programa. Por lo tanto, la estrategia para utilizar este recurso siempre consiste en los mismos pasos:

1. Determinar el tamaño del “bloque” de datos necesario
2. Crear o, más precisamente, “reservar”[18] el espacio de datos usando las funciones existentes.

[18] La memoria de la computadora, por supuesto, ya existe antes de que se ejecute el programa, por eso en lugar de hablar de “crear” un espacio de memoria, que es lo que quizás percibimos al programar, hablamos de “reservar” un espacio. Se toma un espacio existente para utilizarlo en el programa.

3. Asignar una referencia a ese bloque de memoria para manipularlo.
4. Usar la referencia para escribir y leer datos del bloque
5. Liberar el espacio al terminar.

Para varias de estas tareas usaremos las herramientas que ya conocemos de C, pero para reservar y liberar memoria es imprescindible utilizar las funciones de la librería estándar `malloc()` (cuyo nombre proviene de "**m**emory **alloc**ation", que en español significa "asignación de memoria") y `free()` que, como su nombre en inglés lo indica, "libera" el recurso.

Recordemos que luego de declarar una variable o un arreglo, utilizamos el nombre de la variable para guardar y recuperar información en ella. En cambio, cuando reservamos espacio utilizando `malloc()` ese espacio quedaría aislado y sin ninguna forma de ser accedido, si no fuera porque `malloc()` retorna una referencia a este recurso. Al llamar a esta función, por lo tanto, debe almacenarse la referencia que retorna en un puntero y este puntero será la herramienta que se utilizará para leer o almacenar información.

Los códigos 9.2 y 9.3 muestran dos formas de almacenar una serie de datos. El código 9.2 lo hace a través de una declaración de un arreglo "estático" mientras que el código 9.3 utiliza la herramienta de memoria dinámica.

```c
#include <stdio.h>

int main()
{
   int i,n;
   int arr[50];
   printf("Cantidad de elementos a ingresar: ");
   scanf("%d",&n);
   for(i=0;i<n;i++){
      scanf("%d",&arr[i]);
   }
   printf("Elementos ingresados: ");
   for(i=0;i<n;i++){
      printf("\n%d",arr[i]);
   }
   return 0;
}
```

Código 9.2. *Se almacena una serie de datos en un arreglo estático declarado con un tamaño que se supone suficiente para la cantidad de datos a ingresar.*

```c
#include <stdio.h>
#include <stdlib.h>
int main()
{
   int i,n;
   int *p;
   printf("Cantidad de elementos a ingresar: ");
```

```
8       scanf("%d",&n);
9       p = (int*)malloc(sizeof(int)*n);
10      for(i=0;i<n;i++){
11         scanf("%d",(p+i));
12      }
13      printf("Elementos ingresados: ");
14      for(i=0;i<n;i++){
15         printf("\n%d",*(p+i));
16      }
17      free(p);
18      return 0;
19   }
```

Código 9.3. *Se almacena una serie de datos consultando previamente cuántos serán y reservando el espacio en memoria necesario para esa cantidad exacta de datos.*

En el código 9.2 se reserva espacio para 50 enteros y se espera el ingreso de un tamaño que dictará cuántos de esos valores se usarán realmente[19]. Luego, ese valor se utiliza como límite en un bucle para leer los datos y almacenarlos en el arreglo `arr[]`. Finalmente, se utiliza un procedimiento análogo para imprimirlos en pantalla.

En el código 9.3 se realiza un procedimiento similar, haciendo uso de memoria dinámica. La primera diferencia notable es la inclusión de la librería `stdlib.h`, que contiene las definiciones de las funciones necesarias.

La segunda diferencia importante es que no se declara ningún arreglo. En la línea 6 del código 9.2 se declaraba el arreglo `arr[]` de 50 elementos enteros, es decir, para una computadora donde cada entero ocupa 4 bytes, se declara un arreglo de un tamaño total de 200 bytes. En el código 9.3 no se declara un arreglo, pero se sabe que se reservará memoria para uno más adelante y por lo tanto se declara un puntero `p`, previendo la necesidad de almacenar la referencia a ese bloque de memoria que se obtendrá luego.

En las líneas 7 y 8 de ambos códigos, se pide y lee por teclado un tamaño que se almacena en la variable `n`. En el código 9.2 este valor establecerá el límite para recorrer `arr[]`, por lo que de los 200 bytes que se reservaron sólo se utilizarán $n \times 4$ (nuevamente considerando que el tamaño del entero es 4 bytes, y que n es menor a 50). En el código 9.3 en cambio, ese valor se utiliza directamente para calcular el espacio necesario y se reserva ese espacio con la función `malloc()`, todo en la línea 9.

En la figura 9.1 se analiza la línea 9 en mayor detalle, y pueden observarse las múltiples acciones que allí se realizan.

[19] Tanto en el código 9.2 como en el 9.3 se omiten, por simplicidad, verificaciones que en un programa real serían importantes para que no se ingrese un valor que supere el máximo tamaño disponible en el arreglo.

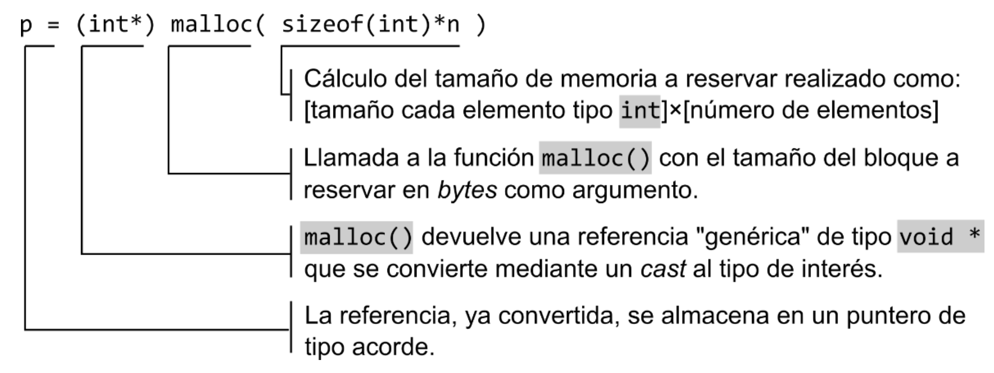


Figura 9.1. *Análisis de la línea 9 del código 9.3 ejemplificando el uso de la función `malloc()`.*

Luego de reservar el bloque de memoria del tamaño necesario para almacenar `n` enteros, las siguientes líneas son similares en los códigos 9.2 y 9.3, pero mientras en el 9.2 se utiliza el nombre del arreglo, en el 9.3 se accede a las posiciones utilizando el puntero `p` que contiene la referencia al bloque.

Finalmente, en la línea 17 del código 9.3 se observa que al finalizar la utilización del bloque reservado, se libera el recurso por medio de la función `free()` con argumento `p` para identificar el bloque a liberar.

Funciones de manejo de memoria dinámica

Todas las operaciones específicas de memoria dinámica se realizan con unas pocas funciones de la librería estándar que se detallarán a continuación. En particular, se describirán las 4 funciones más importantes. Más adelante se estudiarán estrategias más complejas para usar memoria dinámica, como las “listas enlazadas”, pero estas estrategias no son más que algoritmos que siempre hacen uso de estas pocas funciones.

Función de asignación de memoria

En primer lugar, se completará el análisis, comenzado en la sección anterior, de la función que permite reservar espacio en memoria, cuyo prototipo es el siguiente:

```
void *malloc(size_t size)
```

El único dato que necesita es el tamaño en bytes del bloque a reservar, que está indicado en este prototipo por la variable `size`. El número de bytes se indica siempre con un entero positivo, pero según la computadora el tamaño máximo de bytes disponibles puede variar; por ese motivo en lugar de definirse este parámetro como `unsigned int` que podría limitar el tamaño en algunos sistemas se usa el tipo `size_t` que puede representar un `unsigned int`, un `unsigned long long`, o el tipo que mejor se adapte al sistema de cómputo para el cual se compila. Sin embargo, al programar sólo debemos preocuparnos por pasar como argumento un número entero positivo y la conversión se realizará implícitamente (siempre y cuando no se exceda el rango).

Conociendo el número de bytes que debe reservar, `malloc()` intenta asignar un bloque del tamaño pedido para su uso en el programa, y devuelve una referencia al bloque para que pueda accederse al mismo. Si falla en el intento de reservar memoria, la referencia que retorna tendrá el valor `NULL`. Esta característica es la que permite verificar si el bloque de memoria pedido realmente estaba disponible y tomar acciones correctivas en caso contrario.

Esta función puede usarse para reservar tipos de datos de tanta diversidad como quien programa pueda imaginar, incluyendo arreglos de estructuras (es decir, un tipo definido por quien programa y por lo tanto desconocido para la librería estándar). En consecuencia, `malloc()` devuelve una referencia de tipo `void *` que puede verse como un puntero genérico que no conoce a qué tipo de dato apunta. Por supuesto, si se quiere desreferenciar el puntero para acceder a la información, debe conocerse el tipo; recordemos la problemática planteada en el capítulo de punteros: cuando se mira una dirección de memoria, la información almacenada depende de si se observa 1 sólo byte, o 2, o 4, y si se interpretan como carácter, entero, real, etc. Por lo tanto, el puntero debe convertirse al tipo deseado con una operación de *cast*.

Función de asignación e inicialización a 0 de memoria

Otra función que sirve para la reserva de memoria es `calloc()`. Es muy similar a `malloc()` pero inicializa toda la memoria reservada con ceros. El prototipo es el siguiente:

```
void *calloc(size_t num, size_t size)
```

Además de inicializar la memoria con ceros, hay una diferencia en la forma de indicar los argumentos: mientras `malloc()` tomaba un solo argumento entero que indicaba el número total de bytes, `calloc()` recibe dos argumentos enteros para indicar el número n de bloques y el tamaño t de cada bloque, para reservar $n \times t$ bytes. Son dos maneras distintas de especificar lo mismo.

El código 9.4 muestra un ejemplo del uso de `calloc()` realizando un programa que cuenta los caracteres presentes en cada línea o renglón de un archivo de texto.

```
FILE *pf;
int i=0,lineas=,*arr_carac;
// Apertura del archivo y recuento de líneas
pf = fopen("texto.txt","r");
while(!feof(pf)){
    if(fgetc(pf)=='\n')
        lineas++;
}
// Reserva del arreglo de contadores
arr_carac = (int*)calloc(lineas,sizeof(int));
if(arr_carac==NULL){
    printf("Error. Fin del programa");
    exit(1);
}
// Conteo de caracteres por línea
rewind(pf);
while(!feof(pf)){
```

```
    if(fgetc(pf)!='\n')
        arr_carac[i]++;
    else
        i++;
}
fclose(pf);
// Impresión de resultados
for(i=0;i<lineas;i++){
    printf("%d\n",arr_carac[i]);
}
free(arr_carac);
return 0;
```

Código 9.4. *Ejemplo del uso de la función* `calloc()`.

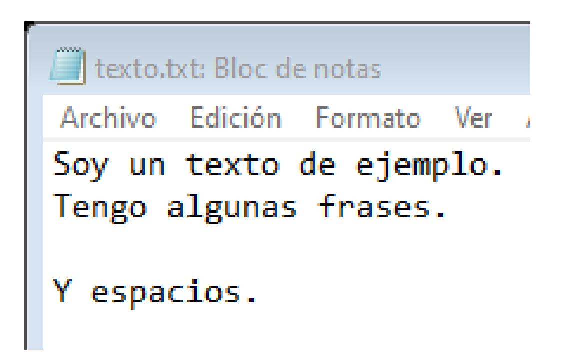


Figura 9.2. *Archivo de entrada para el código 9.4*

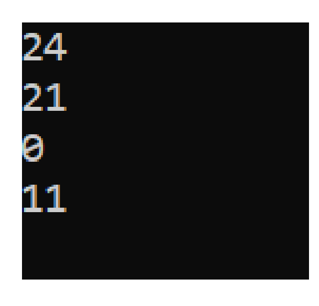


Figura 9.3. *Salida del programa del código 9.4 al ingresarse el archivo de la figura 9.2.*

Se necesita un variable contadora por cada línea de texto, o un arreglo de contadores de la dimensión apropiada, pero no se conoce de antemano cuántas líneas tiene el archivo; esto sólo puede conocerse en tiempo de ejecución al abrir y analizar el archivo. Por lo tanto, se recurre a la estrategia de memoria dinámica para reservar este arreglo. Se deja como ejercicio el análisis completo del código 9.4 ya que comprende temas vistos en capítulos anteriores, pero se remarca el uso de la función `calloc()` para reservar memoria para un arreglo de contadores enteros inicializados en 0. Al mismo tiempo, se hace uso del valor que devuelve `calloc()`, a diferencia del ejemplo del código 9.3 donde se omitió este detalle por simplicidad. El programa demuestra una asociación frecuente entre reserva de memoria dinámica y manejo de archivos, ya que por lo general se desconoce el contenido de los archivos antes de abrirlos.

Función de reasignación de memoria

La tercera función de importancia es `realloc()`, que como su nombre sugiere *reasigna* un espacio previamente reservado modificando su tamaño. El prototipo es el siguiente:

```
void *realloc(void *p, size_t size)
```

Los argumentos de la función son dos: el bloque a reasignar y el nuevo tamaño. La función conserva todos los datos que existían previamente en el bloque de memoria y devuelve una

referencia al nuevo espacio. Si bien puede imaginarse que la función “redimensiona” el mismo bloque que se había reservado previamente, en realidad puede también cambiar su ubicación (trasladando los datos) por lo que la referencia que devuelve puede no ser la misma que antes apuntaba al bloque. La referencia devuelta será `NULL` también en caso de error. El código 9.5 muestra un breve ejemplo de uso de esta función.

```
float *p;
int i;
// Reserva de un arreglo de 5 floats
p = (float*)malloc(5*sizeof(float));
// Se rellenan los 5 elementos
for(i=0; i<5; i++){
    *(p+i)=i+1;
}
// Reasignación para 10 floats
p = (float*)realloc(p,10*sizeof(float));
// Se rellenan los últimos 5 elementos
for(i=5; i<10; i++){
    *(p+i)=i+1;
}
// Impresión del arreglo completo
for(i=0; i<10; i++){
    printf("%.2f\n",*(p+i));
}
free(p);
```

Código 9.5. *Reasignación de un bloque de memoria para almacenar el doble de datos que lo originalmente reservado.*

```
1.00
2.00
3.00
4.00
5.00
6.00
7.00
8.00
9.00
10.00
```

Figura 9.4. *Salida por pantalla del código 9.5*

El tamaño indicado a `realloc()` puede ser mayor o menor al previamente asignado. Incluso puede asignarse un tamaño 0 y el efecto será el mismo que el de la última función que detallaremos a continuación.

Función de liberación de memoria

La función `free()`, “libera” un bloque previamente reservado.

`void free(void *p)`

Esta función no devuelve nada, y toma como argumento la referencia al bloque de memoria que se quiere liberar. Debe llamarse a `free()` por cada bloque que se haya reservado luego de terminar de usarlo, antes de finalizar el programa. De no hacerlo, se incurrirá en lo que se conoce en inglés como “*memory leak*” o fuga de memoria. Por esta razón, debe cuidarse de no perder la asociación entre el puntero y el bloque hasta no liberar el recurso. Por ejemplo, si se sobrescribe el puntero con la dirección de un segundo bloque, el primero quedará inaccesible y nunca podría liberarse.

Arreglos, matrices y estructuras utilizando memoria dinámica

Arreglos unidimensionales

Los arreglos “simples” son la primera estructura de datos que naturalmente surge al trabajar con memoria dinámica, como se vio en los ejemplos de las secciones anteriores. Antes de continuar con elementos más complejos se verá, en el código 9.6, un ejemplo que utiliza `malloc()` y `realloc()` para manipular un “arreglo de arreglos”.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
char* leer_cadena(void);
int main()
{
    int i,total = 0;
    char op;
    char **arr;

    printf("Ingrese los nombres deseados.\n");
    printf("Al terminar, se imprimen en pantalla.\n");
    arr = (char**)malloc(sizeof(char*));
    do{
        arr = (char**)realloc(arr,(total+1)*sizeof(char*));
        if(arr==NULL){
            printf("Error de reserva de memoria.");
            exit(1);
        }
        printf("Ingrese un nombre: ");
        *(arr+total)  = leer_cadena();
        total++;
        printf("Desea ingresar otro? s/n ");
        op=getchar();
        fflush(stdin);
    }while(op!='n');

    for(i=0; i<total; i++)
    {
        printf("%s\n",arr[i]);
        free(arr[i]);
    }
    free(arr);
    return 0;
}

char *leer_cadena(){
    char aux[100];
    char *p;
    int l;
    scanf("%99[^\n]",aux);
```

```
        fflush(stdin);
        l = strlen(aux)+1;
        p=(char*)malloc(l*sizeof(char));
        if(p!=NULL){
            strcpy(p,aux);
        }
        return p;
    }
```

Código 9.6. *Se reserva memoria en forma dinámica para almacenar cadenas de largo desconocido. A su vez, se reserva dinámicamente memoria para manejar las referencias a cada cadena.*

El objetivo del código 9.6 es almacenar cadenas de caracteres utilizando la menor cantidad de bytes posibles. Por lo tanto, la función `leer_cadena()` reserva el espacio exacto para cada cadena considerando su número de caracteres (más 1, por el carácter nulo). Es interesante destacar que se reserva memoria dentro de la función, pero esa memoria permanece asignada durante la ejecución de todo el programa y se libera sólo al terminar.

Al generar las cadenas dinámicamente y pretender mantener todas las cadenas en uso durante la ejecución, debe también procurarse algún mecanismo para almacenar las referencias a cada una de esas cadenas; para eso se utiliza otro arreglo apuntado a su vez por la variable `arr`. Remarquemos que `arr` apuntará a un arreglo de referencias a cadenas de caracteres, es decir, que puede verse como un arreglo de punteros a caracteres. Por lo tanto, la referencia para manejar `arr` será de tipo *puntero a puntero a carácter*, lo cual justifica su declaración como `char **arr` y el *cast* que debe realizarse al reservar memoria: `(char**)`.

Debido a que el número de cadenas se incrementa si el usuario así lo desea, el número de referencias a almacenar también se incrementa y por lo tanto debe modificarse el tamaño del arreglo apuntado por `arr` cada vez que se añade una nueva cadena, como ejemplifica la Figura 9.5.

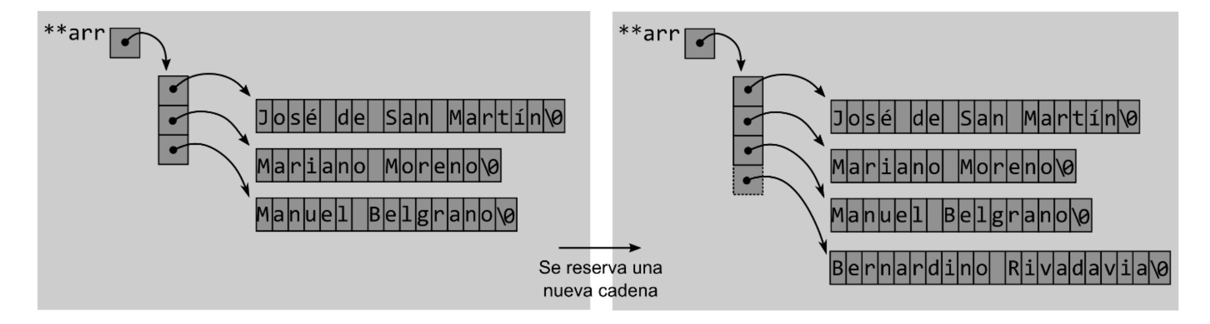


Figura 9.5. *Ejemplo gráfico de la reasignación del arreglo de referencias en el código 9.6*

En consecuencia, el arreglo de referencias no sólo debe ser reservado en forma dinámica, sino que debe reasignarse cada vez con un tamaño mayor. En el código 9.6 se utiliza la función `realloc()` con este propósito.

Arreglos bidimensionales

Mencionaremos 3 maneras distintas de reservar dinámicamente espacio en memoria para una matriz o arreglo bidimensional. Las 3 conducen al mismo resultado general, pero con distinto grado de esfuerzo, ventajas y desventajas. Ya conocemos la moraleja gracias al cuento de los 3 chanchitos y el lobo feroz: todos terminan construyendo una casa, unos con más rapidez, pero con resultados de mayor fragilidad, y otros en forma más lenta y compleja, pero obteniendo una casa más robusta. Por supuesto, a diferencia de los chanchitos, quien programa quizás sepa que en su bosque no hay lobos y entonces la mejor opción no siempre será la más robusta.

La casa de paja: el arreglo impostor

Recordando lo visto en los capítulos de arreglos y de punteros, cuando se declara un arreglo bidimensional de F filas por C columnas con una sentencia como `int mat[F][C]`, (donde `F` y `C` son constantes) en la memoria de la computadora se reserva espacio colocando una fila de C elementos a continuación de la otra. Esto es, visto como bloque de memoria, indistinguible de un arreglo "largo" de $F \times C$ elementos (se muestra gráficamente en la figura 6.5 del capítulo 6 sobre Punteros). El método propuesto en esta sección es entonces, simplemente, ante la necesidad de utilizar una matriz de F filas por C columnas, reservar espacio para un arreglo de $F \times C$ elementos y utilizarlo *como si fuese* una matriz.

Para usar el arreglo de $F \times C$ elementos como una matriz, se debe hacer un uso inteligente de los índices, considerando que acceder al elemento `mat[i][j]` es equivalente a acceder al elemento número $i \times C + j$ del arreglo. El código 9.7 muestra un ejemplo sencillo declarando una matriz y al mismo tiempo reservando espacio para una matriz de las mismas dimensiones equivalentes, pero con el método propuesto.

```
#include <stdio.h>
#include <stdlib.h>
#define F 3
#define C 5
int main()
{
    int *pmat;
    int i,j,n;
    int mat[F][C];
    pmat = (int*)malloc(F*C*sizeof(int));
    if(pmat==NULL) exit(1);
    n=0;
    for(i=0;i<F;i++){
        for(j=0;j<C;j++){
            mat[i][j] = n;
            *(pmat+i*C+j) = n;
            n++;
        }
```

```
        }
        free(pmat);
        return 0;
    }
```

Código 9.7. *Reserva de un arreglo de tamaño F x C para utilizar como matriz. Al finalizar la ejecución, ambas matrices almacenan los mismos valores.*

La ventaja de este método es que es sumamente sencillo y rápido de implementar. La primera desventaja es que requiere una notación relativamente complicada con el álgebra de los índices para hacer uso de la matriz, lo cual fácilmente conduce a errores. Como desventaja adicional, el uso de un puntero simple no es compatible con funciones que tomen matrices como argumento, por lo que deberá recurrirse a pasos extra para compatibilizarlas. El siguiente método permite resolver este problema.

La casa de madera: Un puntero refinado

El segundo método no siempre puede aplicarse. Se basa en declarar un puntero que es del mismo tipo que el nombre de una matriz tradicional. El mecanismo y la estructura del puntero se explica en la sección de arreglos multidimensionales del capítulo 6; aquí solo recordaremos que para referenciar una matriz de C columnas de tipo t y mantener el álgebra de punteros, se utiliza un puntero del tipo "puntero a arreglo de C elementos de tipo t". Por lo tanto, ese es el puntero que se genera en este método para administrar la memoria reservada, como se visualiza en el ejemplo del código 9.8.

```
#define F 3
#define C 5
int main()
{
    int k=C;
    int (*pmat)[C];  // puntero a arreglo de C enteros
    int i,j,n;

    int mat[F][C];
    pmat = (int (*)[k] )malloc(F*C*sizeof(int));

    n=0
    for(i=0;i<F;i++){
        for(j=0;j<C;j++){
            mat[i][j] = n;
            pmat[i][j] = n;
            n++;
        }
    }
    free(pmat);
    return 0;
}
```

Código 9.8. *Declaración de un puntero para reservar una matriz dinámica conservando la notación de índices y la compatibilidad con funciones con matrices como argumentos.*

La primera desventaja importante de este método es que para declarar el puntero debe conocerse el número de columnas C, lo cual va en contra del concepto de reserva dinámica. Sin embargo, en muchos casos puede ser suficientemente flexible, o puede usarse dentro de una función a la que se le pasa como argumento el valor de C.

El segundo problema, que este método comparte con el primero, es que se reserva todo el bloque completo de memoria de una sola vez, y si no está disponible, el programa falla. El siguiente método da una solución a esta debilidad.

La casa de ladrillos: Un arreglo de referencias

El tercer método sigue un concepto similar al utilizado en el código 9.6, consiste en reservar memoria para cada fila, es decir para F arreglos independientes de C elementos cada uno. Por supuesto, si se reservan F arreglos deben tenerse también F referencias, lo cual se logra reservando a su vez un arreglo de tamaño F para alojarlas. El código 9.9 muestra este tipo de reserva a través de un programa para ingresar y calcular el producto vectorial de dos vectores de largo arbitrario.

```
#include <stdio.h>
#include <stdlib.h>

float *ingreso_vector(int n);

int main()
{
    int i,j,n_fil,n_col;
    float *pv1,*pv2,**pmat;

    /* Se ingresan los vectores */
    printf("Ingrese largo del 1er vector: ");
    scanf("%d",&n_fil);
    pv1 = ingreso_vector(n_fil);
    printf("Ingrese largo del 2do vector: ");
    scanf("%d",&n_col);
    pv2 = ingreso_vector(n_col);

    /* Se reserva la matriz */
    pmat = (float**)malloc(n_fil*sizeof(float*));
    for(i=0;i<n_fil;i++){
        pmat[i] = malloc(n_col*sizeof(float));
        if(pmat[i]==NULL) break;
    }
    if(i!= n_fil){
        printf("Se reservaron %d filas solamente.\n",i);
        n_fil=i;
    }

    /* Se realiza el producto */
    for(i=0;i<n_fil;i++){
```

```
        for(j=0;j<n_col;j++){
            pmat[i][j] = pv1[i]*pv2[j];
        }
    }

    /* Se imprime en pantalla */
    for(i=0;i<n_fil;i++){
        for(j=0;j<n_col;j++){
            printf("%f\t",pmat[i][j]);
        }
        printf("\n");
    }

    /* Se liberan los recursos */
    free(pv1);
    free(pv2);

    for(i=0;i<n_fil;i++){
        free(pmat[i]);
    }
    free(pmat);

    return 0;
}

float *ingreso_vector(int n){
    float *pv;
    int i;
    pv =(float *)malloc(n*sizeof(float));
    for(i=0;i<n;i++){
        printf("Ingrese elemento nro %d: ",i+1);
        scanf("%f",&pv[i]);
    }
    return pv;
}
```

Código 9.9. *Demostración de la reserva de una matriz como colección de filas, utilizado para resolver el producto de dos vectores.*

Se sabe que el producto de un arreglo de tamaño n_1 multiplicado por uno de tamaño n_2 dará como resultado una matriz de n_1 filas por n_2 columnas. Por lo tanto en el código 9.9 se asigna la longitud del primer arreglo a la variable `n_fil` y la del segundo a `n_col`. Utilizando estas dimensiones, se reserva primero un arreglo de `n_fil` elementos para almacenar las referencias, y luego se reserva cada una de las filas de tamaño `n_col`. Puede observarse en el código que tras reservar el espacio para la matriz siguiendo este método, es posible utilizar la notación de índices `pmat[i][j]` ya que la aritmética de punteros es compatible con la ordenación implementada de los arreglos. En efecto, las reglas de aritmética de punteros dictan que `x[i]` es exactamente equivalente a `*(x+i)`. Por lo tanto, `pmat[i][j]` es equivalente a `*(pmat[i]+j)`. En esta última expresión, gracias a la reserva efectuada, `pmat[i]` es la referencia a la dirección de la i-ésima

fila, dentro de la cual nos movemos `j` lugares (es decir, a la j-ésima columna) y obtenemos el valor allí almacenado usando el operador `*`.

Es importante notar que al finalizar el uso de la matriz, no se libera simplemente el arreglo apuntado por `pmat` sino que, antes, también se libera *cada una de las filas*.

Este tercer método permite utilizar la notación de doble índice para recorrer la matriz, y también detectar cuando ya no hay lugar para reservar más "filas" de la matriz. El código 9 seguirá funcionando si sólo se reservan algunas filas: sólo calculará el producto para esas filas. Sin embargo, el puntero utilizado de tipo "*puntero a puntero de tipo T*" es distinto del tipo inherente de las variables de matrices declaradas tradicionalmente que es "*puntero a arreglo de C elementos de tipo T*". Si no se realiza la conversión de tipo, el compilador emitirá una advertencia. El ejemplo del código 9.10 muestra una implementación con pasaje a una función.

```c
#include <stdio.h>
#include <stdlib.h>

#define HORAS 24
void imprimir_tabla(int tabla[][HORAS],int d);

int main()
{
    int **pmat,i;
    int dias;
    FILE*pf;

    pf=fopen("arch.txt","rb");
    fseek(pf,0,SEEK_END);
    dias = (ftell(pf)/sizeof(int))/HORAS;
    rewind(pf);

    pmat = (int**)malloc(dias*sizeof(float*));
    for(i=0;i<dias;i++){
        pmat[i] = malloc(HORAS*sizeof(float));
        if(pmat[i]==NULL) break;
    }

    fread(pmat,sizeof(int),dias*HORAS,pf);
    fclose(pf);

    imprimir_tabla((int(*)[HORAS])pmat,dias);

    for(i=0;i<dias;i++){
        free(pmat[i]);
    }
    free(pmat);

    return 0;
}
```

Código 9.10. *Lectura de datos de un archivo a una matriz reservada en forma dinámica, y pasaje de la matriz a una función.*

Estructuras definidas por el usuario

Establecidos los mecanismos básicos de manejo de memoria dinámica, su uso con estructuras de datos no varía salvando la consideración de que cada bloque tendrá ahora el tamaño de una estructura. Así, la reserva de espacio para un arreglo de 10 estructuras como la siguiente:

```
struct producto{
    char nombre[50];
    int stock;
    float precio;
};
```

se realizaría con el siguiente código:

```
struct producto *arr_productos;
arr_productos = (struct producto*)malloc(10*sizeof(struct producto));
```

El uso de arreglos de estructuras reservadas dinámicamente es muy común en el caso de utilizar archivos binarios que almacenan arreglos de estructuras, ya que normalmente no se conoce cuántas estructuras hay almacenadas en un archivo hasta abrirlo en tiempo de ejecución. Recurriendo a una estrategia como la del código 9.10 pueden manejarse datos a partir de investigar primero el archivo y luego reservar la memoria necesaria.

Estructuras dinámicas de datos

Los arreglos y matrices declaradas dinámicamente permiten tener flexibilidad para manipular datos cuyas características son conocidas durante la ejecución de un programa, a partir de entradas de usuario o de archivos. Sin embargo, presentan cierta "rigidez" o ineficiencia a la hora de realizar algunas operaciones. Pensemos en un arreglo de datos donde se quiere insertar un elemento. La operación de "insertar" en un arreglo, conservando el orden del mismo, implica mover todos sus elementos; de la misma manera que al abrochar mal un botón de la camisa, hay que mover trabajosamente a todos un lugar para colocar el que faltaba. No sólo eso, sino que al reservar memoria para los arreglos dinámicamente usualmente se reserva del tamaño exacto; si se quiere agregar uno o más elementos y no se dispone de espacio contiguo, debe reasignarse el arreglo completo en una posición con memoria suficiente, mudar toda la información y destruir el arreglo anterior.

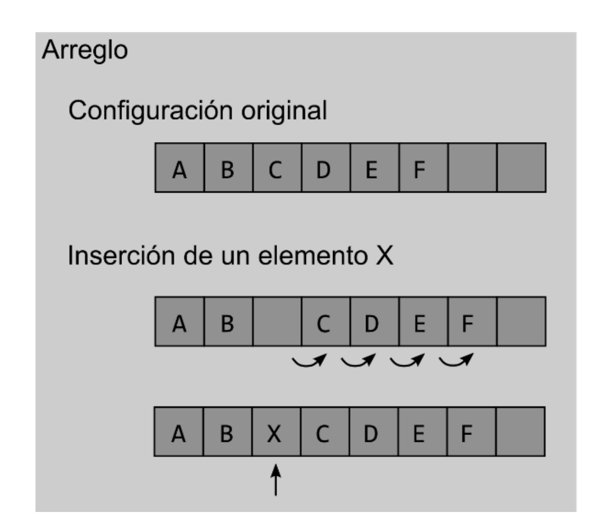
El truco del trencito: Estructuras autoreferenciadas

En lugar de almacenar información en un arreglo con las dificultades que conlleva cuando se busca agilidad en la reconfiguración de los datos, puede recurrirse a otro tipo de organización basada en las llamadas "estructuras autoreferenciadas". El concepto es ilustrado en

la figura 9.6. En contraposición a la configuración rígida impuesta por las “cajas” contiguas de un arreglo, se disponen los datos en lo que podríamos llamar “vagones de tren”, donde cada vagón se enlaza con el siguiente a través de un acople. De esta manera, insertar un elemento entre otros simplemente implica deshacer el acople preexistente y acoplar el nuevo dato, sin mover otro byte de la memoria. Con el método de los “vagones”, para agregar un elemento basta con crear el espacio para él en cualquier posición de memoria, y acoplarlo al resto del “tren”.

Con esta analogía puede analizarse la ventaja del método y también resulta apropiada para señalar la naturaleza de los “acoples” entre vagones. Cada vagón tiene un acople que puede unirse a… otro vagón. Es decir, cada elemento tiene un acople a un elemento *del mismo tipo* y de ahí el nombre de “autorreferenciado”.

Veamos cómo construir cada “vagón” o nodo en lenguaje C. Cada dato debe tener una referencia a otro. Como sabemos, las referencias se implementan con punteros y por lo tanto cada dato deberá estar asociado a un puntero a un dato del mismo tipo. La mejor herramienta para que conviva cualquier dato con un puntero es incluir todo en una estructura que los englobe. El código 9.11 muestra una estructura de este tipo, `struct odc`, con el ejemplo de una “orden de compra”, que incluye datos como cadenas de caracteres (el nombre de un producto), números en punto flotante (el precio), entre otros, y también incorpora un puntero a una estructura de su propio tipo: `struct odc *pnodo` (la autoreferencia).



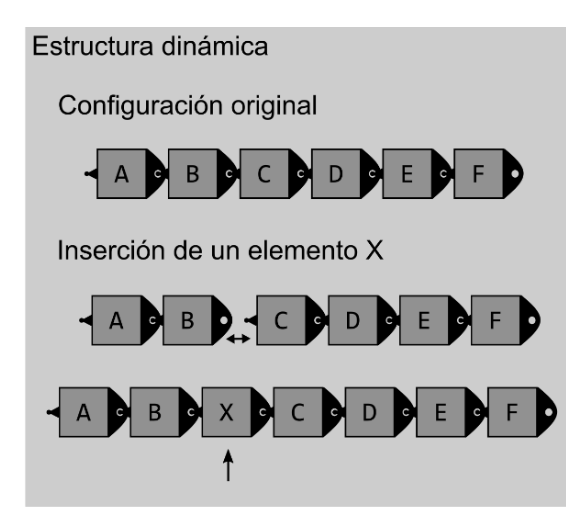


Figura 9.6. *Representación de la inserción de un elemento con dos organizaciones distintas de los datos*

```c
struct odc{
    char producto[50];
    float precio;
    int id_cliente;
    int prioridad;
    struct odc *pnodo;
};
```

Código 9.11. *Ejemplo de una estructura autoreferenciada*

Por supuesto, hay desventajas al utilizar este tipo de estrategia. Más allá de usar más espacio en memoria debido a la necesidad de almacenar un puntero por cada estructura, cada elemento reservado en forma dinámica se ubicará en posiciones de memoria aleatorias, quizás “lejanas” entre sí. Cuando se maneja un gran volumen de datos, recorrer varias de estas estructuras saltando entre posiciones distantes de memoria puede ser muy ineficiente, sobre todo comparado con el caso de los arreglos, cuyos elementos se ubican contiguos en memoria. Por lo tanto, estos métodos no resultarán beneficiosos en aplicaciones como procesamiento numérico y otras donde cada elemento es pequeño y existen muchos elementos que recorrer.

Listas enlazadas simples

La manera más sencilla de organizar información con estructuras autorreferenciads, como se representó en la figura 9.6 y se sugirió con la analogía de un “tren”, es en lo que se conoce como “listas simplemente enlazadas”. El código 9.12 muestra esta estrategia con un ejemplo donde se declaran estructuras autoreferenciadas y se acoplan “a mano”, sin usar memoria dinámica por el momento.

```c
int main()
{
    /* Declaración de elementos de la lista */
    struct odc *inicio = NULL, *fin = NULL;
    struct odc c1 = {"Zapatillas",637.5,25455,3,NULL};
    struct odc c2 = {"Pantalones",450.0,12879,2,NULL};
    struct odc c3 = {"Remera",250.0,8457,3,NULL};

    /* Puntero auxiliar */
    struct odc *paux;

    /* Se forma la lista */
    inicio = &c1;
    c1.pnodo = &c2;
    c2.pnodo = &c3;
    fin = &c3;

    /* Se recorre la lista */
    paux = inicio;
    while(paux != NULL){
        printf("%s\n",paux->producto);
        paux = paux->pnodo;
    }

    return 0;
}
```

Código 9.12. *Ejemplo de estructuras autorreferenciadas configuradas formando una lista. La estructura `struct odc` se define en el código 9.11.*

En el código 9.12 se declaran tres estructuras y, para organizarlas, se van asignando los punteros a los nodos uno a continuación del otro, con las sentencias `c1.pnodo = &c2`. El puntero `pnodo` de una estructura apunta a la siguiente, conformando así la *lista*. En este caso, el último elemento de la lista no apunta a ningún otro elemento, por lo que su puntero se deja con el valor `NULL`.

Además de las estructuras, se declaran punteros específicos como `inicio` y `fin` para conservar en todo momento una referencia a las "puntas" del "tren" que se conforma. Manteniendo estas referencias, puede luego recorrerse la lista saltando de nodo en nodo como se observa al final del mismo código 9.12.

Listas dinámicas y operaciones comunes

El concepto de las estructuras autoreferenciadas y la estrategia de armar listas uniendo los nodos en secuencia entre sí nos ponen en posesión de una herramienta para almacenar y manipular datos en forma muy flexible. Los elementos de la lista pueden crearse y eliminarse en tiempo de ejecución utilizando las herramientas de memoria dinámica vistos en este capítulo. Sin embargo, para mantener la integridad de la lista, debe cuidarse de insertar apropiadamente un elemento creado, o recomponer los acoples al eliminar un elemento.

Insertar un elemento

Imaginemos que se desea agregar un nuevo elemento a la lista conformada en el código 9.11. El nuevo elemento se crea dinámicamente de la siguiente manera:

```
paux = (struct odc*)malloc(sizeof(struct odc));
```

Al crearlo, se referencia momentáneamente con un puntero auxiliar, pero luego se incorporará a la estructura de la lista. Si se quiere insertar al inicio de la lista, los pasos son sencillos:

```
paux->pnodo = inicio;
inicio = paux;
```

En este caso, el nuevo elemento se inserta "entre" el puntero de inicio y la estructura siguiente. El procedimiento sería similar si en lugar de insertarse al inicio, un nuevo elemento N se insertara entre el puntero de una estructura A (en lugar del puntero de inicio) y una estructura B que la siguiese en la lista. Los pasos del procedimiento se ilustran en la figura 9.7 y serían:

(i) El puntero de N se apunta a B, cuya dirección se obtiene del puntero de A.
(ii) Luego, la dirección de N se asigna al puntero de A.

Este procedimiento requiere la modificación de dos punteros. Uno es el de la estructura N, y el otro puede ser el de la estructura A, o puede ser un puntero "libre" como el puntero de inicio de la lista. Estas acciones pueden modularizarse en una función como la del código 9.13. A esta función se pasa una referencia a la estructura nueva (N) para poder modificar su puntero, y una referencia al puntero libre (o el puntero de A, como mencionábamos). Detengámonos un momento en esta última frase: se le pasa una *referencia al puntero*, y no la dirección de memoria

que contiene. Por eso el argumento es `struct odc **posicion`, justamente para poder modificar la dirección de memoria que contiene ese puntero.

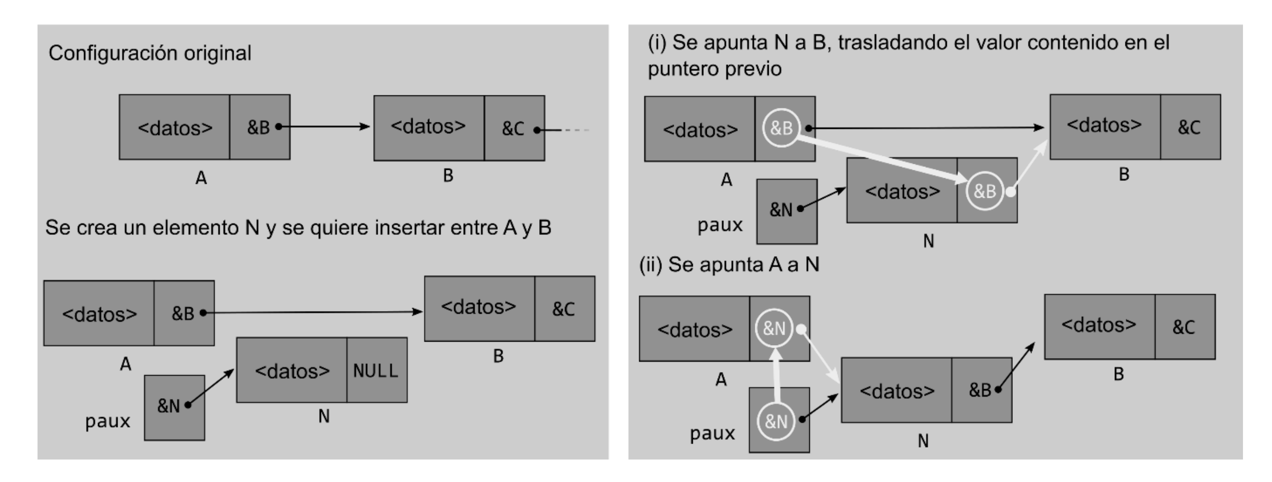


Figura 9.7. *Ilustración del proceso de insertar un elemento entre dos nodos de una lista enlazada. El Elemento A podría ser simplemente el puntero de inicio de la lista.*

```
void insertar(struct odc **posicion, struct odc *nuevo_nodo){
    nuevo_nodo->pnodo = *posicion;
    *posicion = nuevo_nodo;
}
```

Código 9.13. *Función para insertar un elemento en una lista simple. La estructura `struct odc` es una estructura autoreferenciada definida en el código 9.11.*

Si quisiéramos insertar una estructura creada en forma dinámica al comienzo de la lista del código 9.12 utilizando la función del código 9.13, lo haríamos de la siguiente manera:

```
paux = (struct odc*)malloc(sizeof(struct odc));
insertar(&inicio, paux);
```

Y si quisiéramos insertarla entre la estructura `c1` y `c2` usaríamos la sentencia:

```
insertar(&(c1.pnodo), paux);
```

Eliminar un elemento

Otra acción importante para manipular una lista es poder eliminar un elemento de la misma, con el cuidado de liberar el recurso de memoria si fue reservado dinámicamente y de no “romper” la estructura de la lista. Supongamos tener una lista conformada por estructuras como la del código 9.11, cuyos elementos fueron reservados con `malloc()`, y se quiere eliminar la primera de la lista. Eso puede lograrse con las siguientes sentencias:

```
paux = inicio;
inicio = inicio->pnodo;
free(paux);
```

En estas líneas de código se resguarda la dirección del elemento a eliminar en un puntero auxiliar, luego se sobrescribe la dirección que contenía por la del elemento que le sigue (retirándolo de la lista de esa manera) y finalmente se libera el recurso gracias a haber resguardado la

dirección (de otra manera, se hubiese perdido la referencia a ese bloque de memoria al sobrescribir `inicio`). La figura 9.8 ilustra este procedimiento, considerando en forma análoga a la figura 9.7, la eliminación de un elemento en una posición cualquiera de la lista.

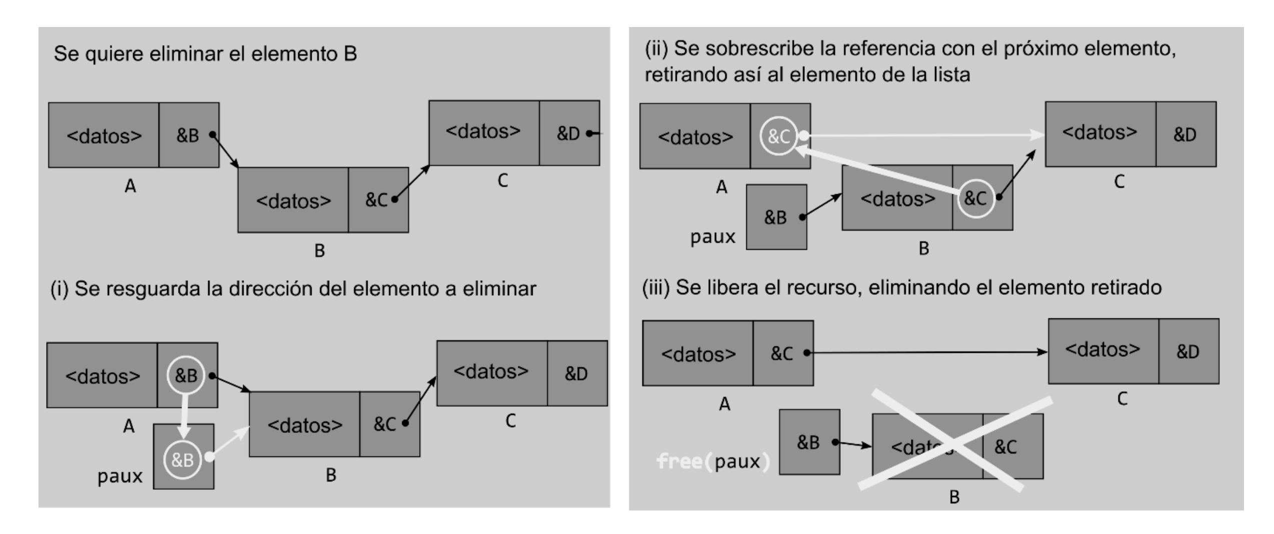


Figura 9.8. *Ilustración del proceso de eliminar un elemento de una lista.*

```
void eliminar(struct odc **p_eliminar){
    struct odc * paux;
    paux = *p_eliminar;
    *p_eliminar = (*p_eliminar)->pnodo;
    free(paux);
}
```

Código 9.14. *Función para insertar un elemento en una lista simple. La estructura struct odc es una estructura autoreferenciada definida en el código 9.11.*

La ilustración de la figura 9.7 sirve de la misma manera para el proceso de eliminar el primer elemento de la lista si se reemplazara el puntero de la estructura A por el puntero de inicio. Por esa razón, en el código 9.14, que demuestra una función para eliminar elementos, el argumento es una referencia a un puntero.

Recorrer la lista

A continuación, se repiten por conveniencia las líneas del código 9.12 que permiten recorrer una lista de principio a fin:

```
paux = inicio;
while(paux != NULL){
    /* Utilizar los elementos a través de paux-> … */
    paux = paux->pnodo;
}
```

El concepto es muy sencillo; se basa en utilizar un puntero auxiliar para saltar de elemento en elemento, utilizando la dirección guardada en el puntero de cada estructura. La última estructura de la lista no apunta a ningún elemento y por lo tanto su puntero contendrá el valor `NULL`, lo cual se utiliza justamente para detectar el final de la lista y detener el bucle.

Tipos específicos de listas enlazadas

A partir de lo visto en las secciones anteriores conocemos la organización general de las listas simplemente enlazadas, y cómo realizar operaciones básicas. Las listas podrían ser usadas de distintas formas para almacenar información y luego acceder a ella, pero es común establecer ciertas reglas de acceso que permiten ordenar los datos y acceder de la manera más eficiente y segura según la aplicación. Según el conjunto de reglas de acceso que se establezca, se dice que las listas son de un tipo u otro.

Pilas o Listas LIFO

Las “pilas” son listas donde se agregan elementos para almacenarlos y se acceden secuencialmente comenzando por el último almacenado. Es análogo a la situación de una pila para lavar los platos después de comer: vamos depositando los platos sucios uno encima del otro al lado de la pileta de la cocina y cuando los queremos lavar, los vamos retirando comenzando por el de más arriba, es decir, el último que se añadió. Por eso se conocen en inglés como LIFO, la sigla de “Last In First Out” que se traduce como “El último en entrar es el primero en salir”.

La estructura de la lista en sí es la misma que se describió en la sección anterior: una lista enlazada simple. Pero, el modo de acceder determina que sea una pila. Por lo tanto, para generar una pila, sólo debe permitirse:

(i) Agregar elementos nuevos al inicio de la lista
(ii) Recuperar o eliminar elementos siempre del inicio de la lista.

Consideremos un programa que administre una pila conformada por estructuras como la del código 9.11 (pero sólo consideraremos los campos `producto` y `precio` por brevedad). Una función que permite escribir un dato en la pila puede ser la propuesta en el código 9.15.

```
void escribir(struct odc **pila, struct odc dato){
    struct odc *paux;
    /* Se crea la nueva estructura */
    paux = malloc(sizeof(struct odc));
    /* Se rellena con los datos a guardar en la pila*/
    strcpy(paux->producto,dato.producto);
    paux->precio = dato.precio;
    /* Se inserta al comienzo de la pila */
    insertar(pila,paux);
}
```

Código 9.15. *Función para guardar un dato en una pila. El dato se suministra como argumento, y se pasa la referencia al puntero de cabecera de la pila. Se hace uso de la función* `insertar()` *del código 9.13.*

Mientras tanto, una función que permite leer un dato de la pila, se observa en el código 9.16. En este caso es importante poder distinguir entre una pila llena y una vacía de alguna manera (al momento de escribir en la pila, en cambio, es irrelevante) por lo que la función devuelve un valor indicando si está vacía o por el contrario tiene algún dato.

```
int leer(struct odc **pila, struct odc *paux){
    // Si la pila está vacía, retorna -1
    if(*pila==NULL){
        return -1;
    }
    // Si no, se copia el elemento correspondiente
    strcpy(paux->producto,(*pila)->producto);
    paux->precio = (*pila)->precio;
    // Y se elimina el elemento del comienzo de la pila
    eliminar(pila);
    return 0;
}
```

Código 9.16. *Función para leer un dato en una pila. El dato se almacena en la estructura referenciada por el segundo argumento, y se pasa la referencia al puntero de cabecera de la pila. Se hace uso de la función* `eliminar()` *del código 9.14.*

Cuando se lee un dato de la pila, el mismo es retirado, por lo que también suele implementarse una función llamada "*peek*" ("espiar" en inglés) que permite obtener los datos del primer elemento de la lista sin eliminarlo.

El código 9.17 muestra un programa completo que hace uso de la pila, aprovechando las funciones definidas en los códigos anteriores.

```
int main()
{
    struct odc *pila = NULL;
    struct odc aux;
    int opcion;
    int resultado;
    do{
        system("cls");
        printf("Presione\n");
        printf("  1 para escribir en la pila.\n");
        printf("  2 para leer de la pila.\n");
        printf("  3 para imprimir la pila.\n");
        printf("  4 para salir  : ");
        scanf("%d",&opcion);
        switch(opcion){
            case 1:
                printf("Ingrese datos: \n");
                printf("Nombre del producto: ");
                scanf("%s",aux.producto);
                printf("Precio: ");
                scanf("%f",&aux.precio);
                escribir(&pila, aux);
                break;
            case 2:
                resultado = leer(&pila, &aux);
                if(resultado==-1){
                    printf("No hay datos en la pila\n");
```

```
                }
                else{
                    printf("Dato: %s, $%.2f\n",aux.producto,aux.pre-
cio);
                }
                break;
            case 3:
                imprimir(pila);
                break;
        }
        printf("\nPresione ENTER para continuar...");
        fflush(stdin);getchar();
    }while(opcion!=4);

    return 0;
}

void imprimir(struct odc * inicio){
    struct odc * paux;
    paux = inicio;
    printf("\n*** Pila ***\n");
    while(paux != NULL){
        printf("%s, $%.2f\n",paux->producto,paux->precio);
        paux = paux->pnodo;
    }
}
```

Código 9.17. *Programa que almacena datos en una pila. Las funciones que utiliza se definen en los códigos anteriores desde el 9.13 al 9.16. Se omiten por brevedad las declaraciones de funciones, inclusión de librerías y la estructura del código 9.11.*

El programa del código 9.17 muestra al usuario un menú con las opciones de escribir un dato en la pila, leerlo, e imprimir la pila completa. Toda la pila se administra con el puntero `struct odc *pila` el cual se inicializa con el valor `NULL` señalando que la pila se encuentra vacía. Como se explicó anteriormente, algunas funciones para administrar la pila necesitan modificar ese puntero, por lo que se les pasa una *referencia* al mismo, en lugar de pasar su valor. En contraste, la función `imprimir()` definida en el propio código 9.17, sólo necesita conocer la dirección del primer elemento de la lista para recorrerla, y por lo tanto recibe el puntero `pila` por valor.

La figura 9.9, al final de la próxima sección, muestra un ejemplo de ejecución del programa donde se ingresan los datos de 3 estructuras y se imprime la pila pudiendo constatarse que el primer elemento ingresado queda en la última posición. Al leerse un elemento, se imprime y retira de la pila el dato que había sido ingresado por último.

Colas o Listas FIFO

Otra manera de organizar las listas es siguiendo la analogía de una cola de espera, como una fila para subir a un colectivo, donde las personas se van agregando al final de la cola y la que

llegó primera es la primera que puede subir al transporte. De ahí el nombre en inglés para estas listas: “FIFO”, las siglas de *First in First out* o “el primero en entrar es el primero en salir”; también se las conoce en inglés como como *Queue* que es la traducción literal de fila o cola.

Para crear una lista como esta puede seguir utilizándose la función `leer()` propuesta en el código 9.16, ya que extrae el primer elemento de la cola. Sin embargo, es necesario repensar la función `escribir()` ya que los elementos deben ser ingresados al final de la lista, en contraste con la pila donde se ingresaban al comienzo. Una estrategia puede ser seguir utilizando un único puntero al inicio de la cola y recorrer los nodos hasta llegar al último, donde se insertará el nuevo elemento. Pero recorrer la lista cada vez no parece ser muy eficiente, sobre todo cuando se almacena un gran volumen de datos, por lo que una mejor estrategia puede ser mantener un segundo puntero al último elemento de la cola.

El código 9.18 muestra los cambios respecto al código 9.17 para que el mismo programa administre la lista como cola (FIFO) en lugar de como pila (LIFO). Todas las observaciones que valían para el código 9.17 también se repiten aquí: se utilizan sólo los campos `producto` y `precio` de `struct odc`, la cual está definida en el código 9.11, y se utilizan las funciones definidas en los códigos 9.14 y 9.16.

```c
/* Agregado a la declaración de variables*/
 struct odc *fin_cola = NULL;

 /* Modificación del algoritmo para la escritura
    de un nuevo elemento */
 case 1:
    printf("Ingrese datos: \n");
    printf("Nombre del producto: ");
    scanf("%s",aux.producto);
    printf("Precio: ");
    scanf("%f",&aux.precio);
    escribir(&fin_cola, aux);
    if(cola==NULL){
        cola=fin_cola;
    }
    break;

/* Modificación de la función escribir()*/
void escribir(struct odc **fin_cola, struct odc dato){
    struct odc *paux;
    /* Se crea la nueva estructura */
    paux = malloc(sizeof(struct odc));
    /* Se rellena con los datos a guardar en la cola*/
    strcpy(paux->producto,dato.producto);
    paux->precio = dato.precio;
    /* Se inserta al final de la cola */
    paux->pnodo = NULL;
    if(*fin_cola != NULL){
        (*fin_cola)->pnodo = paux;
```

```
    }
    *fin_cola = paux;
}
```

Código 9.18. *Segmentos de código a reemplazar en el código 9.17 para crear una cola en lugar de una pila. La diferencia fundamental es que los nuevos elementos se agregan al final de la lista.*

En el código 9.18 se modifican tres segmentos del código 9.17. En primer lugar, se agrega un segundo puntero para administrar la lista: `struct odc* fin_cola`, el cual en conjunción con el puntero `cola` servirá para realizar todas las operaciones necesarias.

En segundo lugar, el algoritmo que se ejecuta cuando el usuario desea añadir un elemento a la cola se modifica porque debe contemplar el caso en el cual la lista está vacía y se añade el primer elemento. Cuando la lista está vacía, los punteros de inicio y fin contienen `NULL`, la lista está “desarmada”. El caso de la pila era más sencillo porque siempre se agregaban elementos al inicio, pero en este caso hay dos punteros y al comenzar el programa están “desincronizados”. Por lo tanto, luego de llamar a la función `escribir` pasándole la referencia al puntero `fin_cola` para que añada allí la nueva estructura, se detecta si es la primera en ser añadida (detectada porque el puntero `cola` contiene `NULL`) y en ese caso el puntero de inicio, `cola`, también se apunta a la misma estructura. Después de todo, si la lista tiene un solo elemento, ese elemento es tanto el principio como el fin. A partir de allí, la lista queda “armada” y puede continuarse con la operatoria normal.

En tercer y último lugar, se modifica la función `escribir()`. Como en la cola se añaden nodos *al final* de la lista, la nueva estructura reservada usando `malloc()` debe contener el valor `NULL` en su puntero, para demarcar el final de la cola. Al mismo tiempo, este nuevo elemento debe ser apuntado tanto por el anterior último elemento como por el puntero `fin_cola`. En consecuencia, el método de inserción es distinto al utilizado en la función `insertar()`, la cual no puede ser aprovechada en este caso.

Con estas tres modificaciones, el mismo código sirve para organizar una lista como cola en lugar de pila. La figura 9.9 muestra la ejecución de ambas versiones del código, ingresando en ambos casos los mismos datos en el mismo orden. Para comparar ambos casos, luego del ingreso de datos, a la izquierda se muestra cómo queda la lista en caso de ser una pila, y a la derecha en el caso de ser una cola.

Ingreso de datos

```
Ingrese datos:
Nombre del producto: Zapatilla
Precio: 2500
```

```
Ingrese datos:
Nombre del producto: Remera
Precio: 700
```

```
Ingrese datos:
Nombre del producto: Buzo
Precio: 1400
```

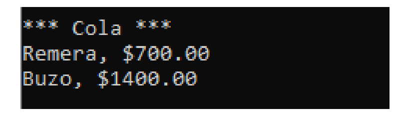
Impresión de la lista

```
*** Pila ***
Buzo, $1400.00
Remera, $700.00
Zapatilla, $2500.00
```

```
*** Cola ***
Zapatilla, $2500.00
Remera, $700.00
Buzo, $1400.00
```

Lectura de un elemento

```
Presione
  1 para escribir en la pila.
  2 para leer de la pila.
  3 para imprimir la pila.
  4 para salir   : 2
Dato: Buzo, $1400.00
```



```
Presione
  1 para escribir en la cola.
  2 para leer de la cola.
  3 para imprimir la cola.
  4 para salir   : 2
Dato: Zapatilla, $2500.00
```

Impresión de la lista tras la lectura

```
*** Pila ***
Remera, $700.00
Zapatilla, $2500.00
```

```
*** Cola ***
Remera, $700.00
Buzo, $1400.00
```

Figura 9.9. *Ejemplo de ejecución del código 9.17 que maneja una pila (centro e izquierda), y el código 9.18 que maneja una cola (centro y derecha). Los tres primeros cuadros al centro son compartidos, se ingresan los mismos elementos en el mismo orden en los dos códigos para comprar la operatoria de la pila y la cola.*

Ejercicios

14) Cree un programa que sea capaz de reservar memoria para un arreglo de enteros del tamaño introducido por el usuario por teclado, que luego se rellene de valores y se imprima en pantalla. No olvide liberar la memoria antes de finalizar el programa.

15) Complete las verificaciones faltantes en el código 3 para asegurar que se reserva memoria exitosamente, y tomar las acciones correctivas pertinentes en caso de falla.

16) Escriba un programa que lea una matriz de un archivo de texto. La matriz se encuentra escrita en el archivo con número reales separados por espacios para demarcar las columnas y con un avance de línea entre filas, y puede ser de cualquier tamaño. El programa debe ofrecer las opciones de imprimir la matriz en pantalla, crear su traspuesta, expandir una fila y rellenarla, expandir una columna y rellenarla, y guardarla nuevamente en otro archivo.

17) Agregue a los programas de los códigos 9.17 y 9.18:
    a) La posibilidad de leer de la pila o cola una cantidad de elementos especificados por el usuario (en lugar de uno sólo).
    b) La función "*peek*" que permite ver el contenido del elemento a leer en una pila o cola sin eliminarlo.
    c) La posibilidad de eliminar un elemento de una posición específica.
    d) La funcionalidad de guardar la lista en un archivo binario o cargar la lista desde un archivo binario.

18) La matriz que modela una orden de compra en el código 9.11 tiene un campo "prioridad". Cree una función que reordene los elementos de una lista de estas estructuras según el valor de este campo. Puede incluir esta funcionalidad en los códigos 9.17 o 9.18.

CAPÍTULO 10
Introducción a la Programación Orientada a Objetos

Pablo A. García

En este capítulo se pretende introducir al lector en un nuevo paradigma de programación, la programación orientada a objetos (POO). En este sentido, se considera importante que el alumno logre modelar los datos que intervienen en los programas haciendo uso de estas nuevas entidades que forman la base del paradigma: los objetos.

Hasta este momento, a lo largo del curso, el alumno ha programado usando el paradigma de programación procedural, haciendo uso de los módulos y las estructuras de programación. En los problemas más avanzados del curso, frente a un determinado programa a implementar, el alumno recurre a las estructuras compuestas para modelar los datos del algoritmo, e implementa módulos con sus interfaces bien definidas.

Luego de presentar una introducción a los conceptos fundamentales de la POO y con el objetivo de diferenciar ambos paradigmas de programación, se presenta un ejemplo integrador de todos los conceptos presentados a lo largo del curso usando los dos paradigmas.

Orden del capítulo

En principio se presenta una introducción a la POO presentando: objetos, clases, atributos, métodos, constructores, sobrecarga, etc. En segunda instancia se presentan los requerimientos de un ejemplo integrador a resolver, invitando al lector a realizar el análisis de la solución procedural, y por último implementar una solución al mismo problema, pero usando el nuevo paradigma de POO.

Introducción a C++

El lenguaje C++ se comenzó a desarrollar en 1980. Su autor fue B. Stroustrup. Al comienzo era una extensión del lenguaje C que fue denominada C con clases. Sus orígenes fueron en la ATT y comenzó a ser utilizado fuera de la misma en 1983. Ante la gran difusión y éxito que iba

obteniendo en el mundo de los programadores, se comenzó a estandarizar. En 1989 se formó un comité para estandarizar a nivel americano e internacional.

En la actualidad, el C++ es un lenguaje versátil, potente y general. Su éxito entre los programadores profesionales le ha llevado a ocupar un puesto importante como herramienta de desarrollo de aplicaciones. Mantiene las ventajas del C en cuanto a riqueza de operadores y expresiones, flexibilidad, concisión y eficiencia.

El C++ es a la vez un lenguaje procedural (orientado a algoritmos) y orientado a objetos. Como lenguaje procedural se asemeja al C y es compatible con él, aunque presenta algunas ventajas. Como lenguaje orientado a objetos se basa en una filosofía completamente diferente, que exige del programador un completo cambio de mentalidad.

Introducción a POO

Es un paradigma de programación que propone resolver problemas identificando objetos y estableciendo relaciones de colaboración entre ellos. En la figura 10.1 se presenta un esquema donde pueden observarse las diferencias entre ambos paradigmas de programación.

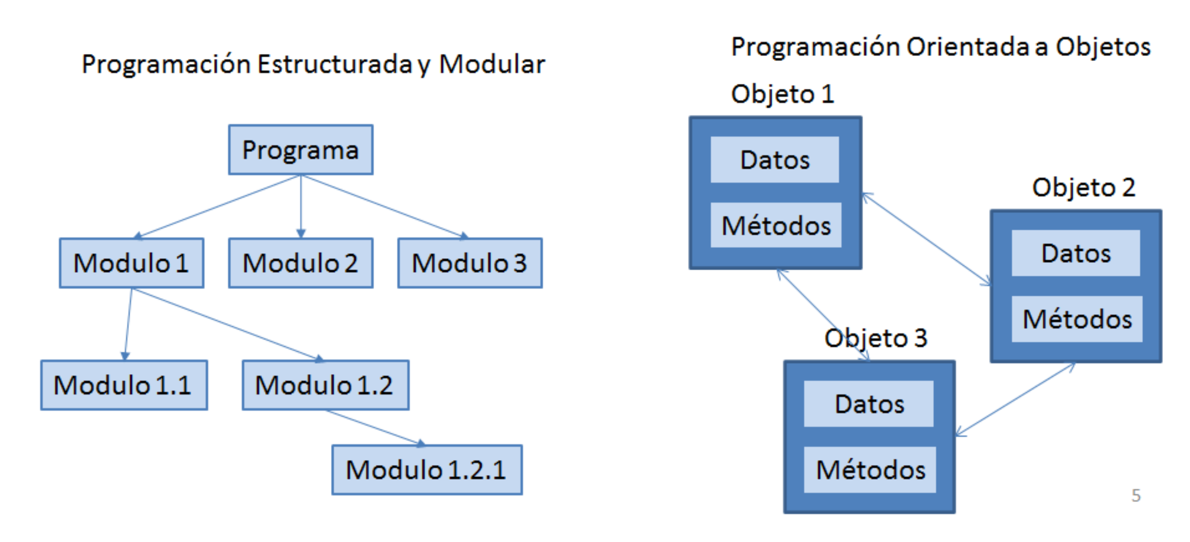


Figura 10.1

La Programación Orientada a Objetos (POO) permite realizar grandes programas mediante la unión de elementos más simples, que pueden ser diseñados y comprobados de manera independiente del programa que va a usarlos. Muchos de estos elementos podrán ser reutilizados en otros programas. A estas “piezas”, “módulos” o "componentes", que interactúan entre sí cuando se ejecuta un programa, se les denomina objetos. Estos objetos contienen tanto datos como las funciones que actúan sobre esos datos, que se conocen como métodos.

Durante la ejecución del programa, los objetos interactúan pasándose mensajes y respuestas.

Un objeto no necesita conocer el funcionamiento interno de los demás objetos para poder interactuar con ellos, sino que le es suficiente con saber la forma en que debe enviarle sus mensajes y cómo va a recibir la respuesta.

La definición genérica de estos objetos se realiza mediante la clase. Así, una clase contiene una completa y detallada descripción de la información y las funciones que contendrá cada objeto de esa clase. Las clases de C++ se pueden ver como una generalización de las estructuras de C.

En ANSI C las funciones son algo relativamente independiente de las variables, y constituyen el centro del lenguaje. Se dice por eso que C es un lenguaje algorítmico (o procedural, en inglés).

Cualquier función se puede comunicar con las demás a través de variables globales, del valor de retorno y de los argumentos, pasados por valor o por referencia. Esta facilidad para comunicarse con otras funciones hace que se puedan producir efectos laterales no deseados.

En un Lenguaje Orientado a Objetos tal como el C++, el centro del lenguaje no son las funciones sino los datos, o más bien los objetos, que contienen datos y funciones concretas que permiten manipularlos y trabajar sobre ellos. Esto hace que la mentalidad con la que se aborda la realización de un programa tenga que ser muy diferente.

Para proteger a las variables de modificaciones no deseadas se introduce el concepto de encapsulación, ocultamiento o abstracción de datos. Los miembros de una clase se pueden dividir en públicos y privados.

La clase ofrece un conjunto de funciones públicas a través de las cuales se puede actuar sobre los datos, que serán privados. Estas funciones o métodos públicos constituyen la interface de la clase. Al usuario le es suficiente con saber cómo comunicarse con un objeto, pero no tiene por qué conocer el funcionamiento interno del mismo.

Existen también las funciones sobrecargadas, que son funciones con el mismo nombre, pero con distintos argumentos y definición.

Otra posibilidad interesante es la de que objetos de distintas clases respondan de manera análoga al aplicarles funciones con idéntico nombre y argumentos. Esta posibilidad da origen a las funciones virtuales y al polimorfismo.

POO: antecedentes

Los conceptos de la POO tienen origen en Simula 67, un lenguaje diseñado para hacer simulaciones, creado por Ole Johan Dahl y Kristen Nygaard, del Centro de Cómputo Noruego en Oslo.

Mejorado en los 70 por Smalltalk desarrollado en Simula en Xerox PARC.

La POO se fue convirtiendo en el estilo de programación dominante a mediados de los años 1980 en gran parte debido a la influencia de C++, se consolidó con el auge de las interfaces gráficas de usuario dando lugar a la programación orientada a eventos.

En el inicio de los 90's se consolida la Orientación a Objetos como una de las mejores maneras para resolver problemas. Aumenta la necesidad de generar prototipos más rápidamente (concepto RAD Rapid Application Developments).

En 1996 surge un desarrollo llamado JAVA y en 2000 surge C# para .NET.

Bits--->POO

- Programación lenguaje máquina 10010110
- Programación hexadecimal 1A 01 04
- Programación assembler MOV AX, 04
- Programación estructurada (código espagueti) GOTO
- Programación modular, se comienzan a encapsular variables locales y a dividir en módulos independientes que se compilan separadamente (métodos).
- Pero para programas importantes llevar la cuenta el programador de todas las variables y todos los métodos resultaba caótico.
- Surge la "La crisis del software", ganando terreno la POO. Se modela usando objetos.

¿Qué es un Objeto?

Uno puede mirar a su alrededor y ver muchos objetos del mundo real: un termo, un mate, una silla, un auto, una computadora, etc.

Cada uno de los objetos comparten dos características:

- Estado
- Comportamiento

Los sustantivos son un buen punto de partida para determinar los objetos de un sistema.

¿Qué es una Clase?

Una clase es una construcción estática que describe un comportamiento común y atributos que toman distintos estados. Su formalización es a través de una estructura de datos que incluye datos y funciones, llamadas métodos. Los métodos son los que definen el comportamiento y los datos describen el estado.

Las clases son declaraciones de objetos. Esto quiere decir que la definición de un objeto es la Clase.

En la figura 10.2 se presenta un esquema detallado donde pueden observarse las diferencias entre clases y objetos.

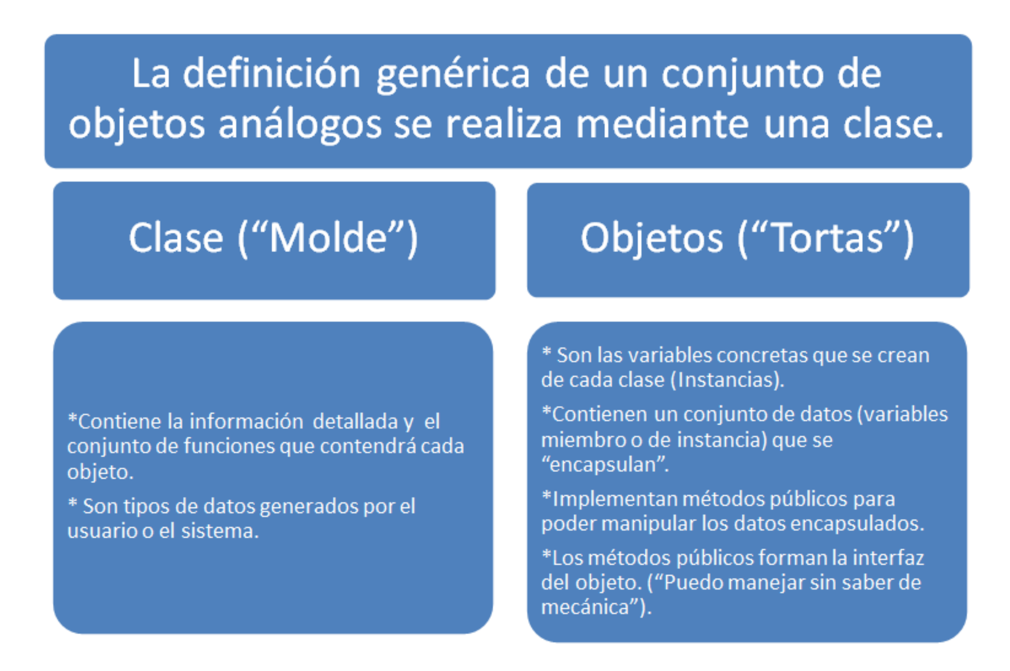


Figura 10.2

Ejemplo de Clase

Implementamos nuestra primera clase celular. ¿Cuál es el estado y el comportamiento que tienen en común todos los teléfonos celulares? En la figura 10.3 se presenta estado y comportamiento a implementar en nuestra clase celular. Es de destacar que el modelo implementado por la clase puede ser muy complejo o simple dependiendo de las necesidades del programador.

Estado	Comportamiento
Marca	Hacer llamada
Modelo	Responder llamada
Número de línea	Enviar SMS
Sistema Operativo	Responder SMS
Color	Enviar Mensaje de Wasap

Figura 10.3

¿Cuáles serían los objetos?

Los objetos son las distintas instancias de una determinada clase que podemos generar. Cada una de las instancias tienen los mismos estados (atributos) y el mismo comportamiento

(métodos). Cada instancia tendrá sus propios valores para cada estado. En la figura 10.4 se pueden observar tres objetos (o instancias) de la clase celular que difieren en su estado “color”.

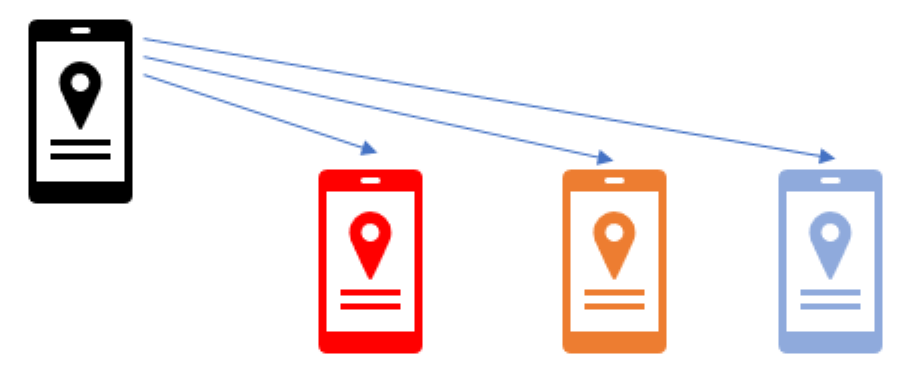


Figura 10.4

Clases: atributos

En la figura 10.5 se puede observar la implementación de la clase celular (líneas 5-11), su instanciación dentro del módulo principal main (línea 17) y el seteo de sus atributos (líneas 18-20).

```
1   #include <iostream>
2
3   using namespace std;
4
5   class celular
6   {
7   public:
8       string marca;
9       string modelo;
10      string color;
11  };
14  int main()
15  {
16      cout << "Creando nuestras primeras clases" << endl;
17      celular tel1;
18      tel1.marca="Samsung";
19      tel1.modelo="S9";
20      tel1.color="Azul";
21      return 0;
22  }
```


Figura 10.5

En la figura 10.6 se puede observar la salida por consola del código que genera dos instancias de la clase celular (tel1 y tel2) imprimiendo en pantalla el estado de sus atributos: marca y modelo.

```cpp
#include <iostream>
using namespace std;
class celular
{
public:
    string marca;
    string modelo;
    string color;
};
int main()
{
    cout << "Creando nuestras primeras clases" << endl;
    celular tel1;
    tel1.marca="Samsung";
    tel1.modelo="S7 Edge";
    tel1.color="Azul";
    celular tel2;
    tel2.marca="Nokia";
    tel2.modelo="1100";
    tel2.color="blanco";
    cout <<"Telefono 1: Marca "<<tel1.marca<<" modelo: "<<tel1.modelo<<endl;
    cout <<"Telefono 2: Marca "<<tel2.marca<<" modelo: "<<tel2.modelo<<endl;
    return 0;
}
```

```
C:\Users\Pablo\Desktop\celular01\bin\Debug\celular01.exe
Creando nuestras primeras clases
Telefono 1: Marca Samsung modelo: S7 Edge
Telefono 2: Marca Nokia modelo: 1100

Process returned 0 (0x0)   execution time : 0.100 s
Press any key to continue.
```

Figura 10.6

Clases: métodos

A las funciones que se implementan en una clase de objetos se las llaman métodos. Dentro de los métodos puede accederse a todos los campos de la clase. Los métodos permiten manipular los datos almacenados en los objetos.

La sintaxis que se usa en C++ para definir los métodos es la siguiente:

```
<tipo devuelto>  <Nombre del método> (<parámetros>)
{
          <instrucciones>
}
```

Por ejemplo: si definimos el método `imprimir()` en nuestra clase celular, nos evitamos de tener que acceder a sus variables de instancia desde el código fuera de la clase.

En la figura 10.7 se implementa el método `imprimir()` dentro de la clase `celular` y luego se utiliza el mismo método desde el `main` para imprimir el estado de las dos instancias generadas: `tel1` y `tel2`.

```
class celular
{
public:
    string marca;
    string modelo;
    string color;

    void imprimir(void)
    {
        cout <<"Telefono: Marca "<<marca<<" modelo: "<<modelo<<endl;
    }
};
int main()
{
    cout << "Creando nuestras primeras clases" << endl;
    celular tel1;
    tel1.marca="Samsung";
    tel1.modelo="S7 Edge";
    tel1.color="Azul";
    celular tel2;
    tel2.marca="Nokia";
    tel2.modelo="1100";
    tel2.color="blanco";
    tel1.imprimir();
    tel2.imprimir();
    return 0;
}
```

Hacemos uso de nuestro método imprimir con ambas instancias

Figura 10.7

Sobrecarga de Métodos

La sobrecarga de métodos en una clase se produce cuando la misma tiene más de un método con el mismo nombre. Las firmas de estos métodos deben ser diferentes.

La firma de un método consiste en:

- El nombre
- El número de parámetros
- El tipo y el orden de los parámetros
- Los modificadores de los parámetros

En el código de la figura 10.8 se puede observar la ampliación en la clase celular con la implementación del método `update_IMEI` sobrecargado tres veces. Las sobrecargas del método se diferencian en los parámetros que recibe: nada en el primer caso, un `unsigned int` en el segundo y una cadena de caracteres en el tercero.

```
class celular
{
public:
    string marca;
    string modelo;
    string color;
    uint16_t IMEI=0;
    void imprimir(void)
    {
        cout <<"Telefono: Marca "<<marca<<" modelo: "<<modelo<<endl;
    }
    uint16_t update_IMEI()
    {
        return(IMEI+=1000);
    }
    uint16_t update_IMEI(uint16_t nuevo_IMEI)
    {
        return(IMEI=nuevo_IMEI);
    }
    uint16_t update_IMEI(char* nuevo_IMEI)
    {
        IMEI=atoi(nuevo_IMEI);
        return(IMEI);
    }
};
```

Agregamos un método update_IMEI con tres sobrecargas en nuestra clase celular

```
int main()
{
    cout << "Creando nuestras primeras clases" << endl;
    celular tel1;
    tel1.marca="Samsung";
    tel1.modelo="S7 Edge";
    tel1.color="Azul";
    celular tel2;
    tel2.marca="Nokia";
    tel2.modelo="1100";
    tel2.color="blanco";
    tel1.imprimir();
    tel2.imprimir();
    cout<<"El nuevo IMEI del telefono 1 es: "<<tel1.update_IMEI()<<endl;
    cout<<"El nuevo IMEI del telefono 2 es: "<<tel2.update_IMEI(1234)<<endl;
    cout<<"El nuevo IMEI del telefono 1 es: "<<tel1.update_IMEI("5678")<<endl;
    return 0;
}
```

```
C:\Users\Pablo\Desktop\celular01\bin\Debug\celular01.exe
Creando nuestras primeras clases
Telefono: Marca Samsung modelo: S7 Edge
Telefono: Marca Nokia modelo: 1100
El nuevo IMEI del telefono 1 es: 1000
El nuevo IMEI del telefono 2 es: 1234
El nuevo IMEI del telefono 1 es: 5678

Process returned 0 (0x0)   execution time : 0.105 s
Press any key to continue.
```

Figura 10.8

Constructor

En el código de la figura 10.9 se puede observar que se generan dos instancias de la clase celular, y que nada impide que se utilice el método imprimir para la instancia `tel2`, aun cuando no se han inicializado sus atributos.

```
int main()
{
    cout << "Creando nuestras primeras clases" <
    celular tel1;
    tel1.marca="Samsung";
    tel1.modelo="S7 Edge";
    tel1.color="Azul";
    celular tel2;
    tel1.imprimir();
    tel2.imprimir();
    return 0;
}
```

Luego de generar una instancia de celular (tel1) asignamos valores a sus campos

Nada impide que usemos el método imprimir para tel2 aún sin haber inicializado sus campos.

```
C:\Users\Pablo\Desktop\celular01\bin\Debug\celular01.exe
Creando nuestras primeras clases
Telefono: Marca Samsung modelo: S7 Edge
Telefono: Marca  modelo:

Process returned 0 (0x0)   execution time : 0.096 s
Press any key to continue.
```

Figura 10.9

La estrategia comúnmente utilizada para asignar campos de un objeto es hacerlo en el momento de su creación a través del pasaje de parámetros. Esto es lo que llamamos Constructor: un método especial que contiene código a ejecutar cada vez que se crea una instancia de esa clase.

La sintaxis de un constructor consiste en definirlo como cualquier otro método, pero dándole el mismo nombre que la clase y no indicando el tipo de valor de retorno.

```
 <nombreClase>(<parámetros>)
{
 <código>
}
```

En el código de la figura 10.10 se puede observar la implementación de un constructor en la clase `celular`. El mismo recibe tres cadenas de caracteres: `mar, mod y col`, que se utilizan para setear los atributos `marca, modelo y color` respectivamente. En el cuerpo del módulo principal se puede observar la forma de invocar al constructor de la clase en el momento de generar las instancias (líneas 42 y 43).

```
class celular
{
public:
    string marca;
    string modelo;
    string color;
    uint16_t IMEI=0;
    celular (string mar, string mod,string col)
    {
        marca=mar;
        modelo=mod;
        color=col;
    }
    void imprimir(void)
    {
        cout <<"Telefono: Marca "<<marca<<" modelo: "<<modelo<<endl;
    }
    uint16_t update_IMEI()
    {
        return(IMEI+=1000);
    }
    uint16_t update_IMEI(uint16_t nuevo_IMEI)
    {
        return(IMEI=nuevo_IMEI);
    }
    uint16_t update_IMEI(char* nuevo_IMEI)
    {
        IMEI=atoi(nuevo_IMEI);
        return(IMEI);
    }
};
int main()
{
    cout << "Creando nuestras primeras clases" << endl;
    celular tel1("Iphone","SE","Plata");
    celular tel2("Huawei","Y2","Blanco");
    tel1.imprimir();
    tel2.imprimir();
    return 0;
}
```

```
C:\Users\Pablo\Desktop\celular01\bin\Debug\celular01.exe
Creando nuestras primeras clases
Telefono: Marca Iphone modelo: SE
Telefono: Marca Huawei modelo: Y2

Process returned 0 (0x0)   execution time : 0.088 s
Press any key to continue.
```

Figura 10.10

Constructor por defecto

En caso de no definir un constructor para la clase el compilador creará uno por defecto:

```
<nombreClase>()
{ }
```

Si definimos un constructor (como en el ejemplo de la figura 10.10), el compilador no incluye ningún otro constructor. Por ello, el intento de instanciación `celular tel3;` (línea 44 de la figura 10.11); da error de compilación pues el constructor por defecto no existe más.

```
int main()
{
    cout << "Creando nuestras primeras clases" << endl;
    celular tel1("Iphone","SE","Plata");
    celular tel2("Huawei","Y2","Blanco");
    celular tel3;
    tel1.imprimir();
    tel2.imprimir();
    return 0;
}
```

Figura 10.11

Para poder utilizar el constructor vacío luego de haber agregado un nuevo constructor con parámetros a nuestra clase, se deberá sobrecargar el constructor, incluyendo el constructor vacío en la sobrecarga.

En el código de la figura 10.12 se puede observar el constructor sobrecargado tres veces, siendo uno de ellos el constructor vacío.

```
class celular
{
public:
    string marca;
    string modelo;
    string color;
    uint16_t IMEI=0;
    celular()
    {
    }
    celular (string mar, string mod,string col)
    {
        marca=mar;
        modelo=mod;
        color=col;
    }
    celular(string mar, string mod,string col,uint16_t IM)
    {
        marca=mar;
        modelo=mod;
        color=col;
        IMEI=IM;
    }
    void imprimir(void)
    {
      cout <<"Telefono: Marca "<<marca<<" modelo: "<<modelo<<" IMEI: "<<IMEI<<endl;
    }
```

Constructor que no recibe parámetros

Constructor que recibe tres cadenas como parámetros

Constructor que recibe tres cadenas y un unsigned int como parámetros

```
int main()
{
    cout << "Creando nuestras primeras clases" << endl;
    celular tel1("Iphone","SE","Plata");
    celular tel2("Huawei","Y2","Blanco",1234);
    celular tel3;
    tel1.imprimir();
    tel2.imprimir();
    tel3.imprimir();
    return 0;
}
```

Ahora podemos utilizar cualquiera de los tres constructores que generamos

```
Creando nuestras primeras clases
Telefono: Marca Iphone modelo: SE IMEI: 0
Telefono: Marca Huawei modelo: Y2 IMEI: 1234
Telefono: Marca  modelo:  IMEI: 0

Process returned 0 (0x0)   execution time : 0.100 s
Press any key to continue.
```

Figura 10.12

Miembros

Tanto los atributos como los métodos de una clase pueden ser:

- de instancia
- de clase

En C++, si queremos declarar miembros de clase usamos la palabra reservada static.

```
static <tipo de dato> unCampoDeClase;
static void UnMetodoDeClase() { }
```

Si queremos que los miembros sean de instancia no ponemos nada, como hemos venido haciendo hasta ahora en nuestros ejemplos previos.

```
<tipo de dato> unCampoDeInstancia;
void UnMetodoDeInstancia() { }
```

Los miembros de instancia, ya sean campos o métodos, se utilizan cuando se trabaja con instancias.

Miembros de instancia

```
int main()
{
    cout << "Creando nuestras primeras clases" << endl;
    celular tel1("Samsung","S7 Edge","Azul");
    celular tel2("Nokia","1100","Blanco");
    celular tel3 ("Motorola","G4","Negro");
    cout<<tel1.marca<<endl;
    cout<<tel2.marca<<endl;
    cout<<tel3.marca<<endl;
    return 0;
}
```

La clase celular define una única variable marca. ¿Por qué puede almacenar tres valores distintos?

```
Creando nuestras primeras clases
Samsung
Nokia
Motorola

Process returned 0 (0x0)   execution time : 0.147 s
Press any key to continue.
```

Figura 10.13

Como se puede observar en el código de la figura 10.13, se crean 3 instancias de la clase celular. Para cada una de ellas, el compilador crea una nueva copia en memoria de la clase.

Hay tres variables marca, tres variables modelo y tres variables color (figura 10.14).

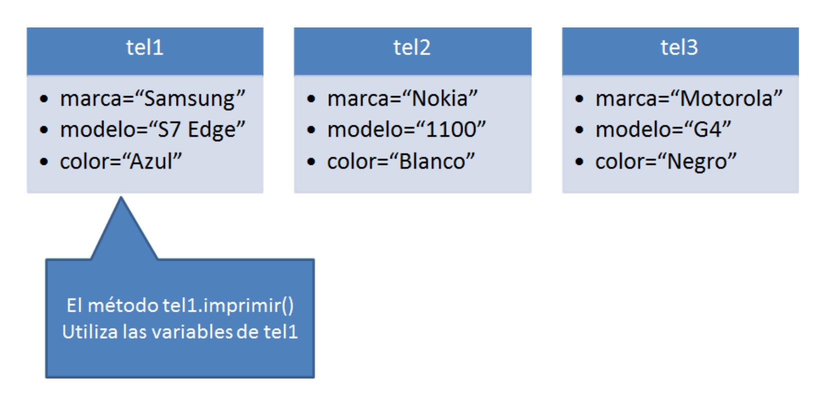


Figura 10.14

Los métodos de instancia son los que acceden a cada variable dependiendo de quién sea el objeto receptor del mensaje.

Miembros de clase

En el código que se muestra en la figura 10.15 se puede observar la clase celular con un atributo de clase (estático) agregado. Como se puede observar en el código, éste atributo “`impresiones`” incluye el modificador `static`.

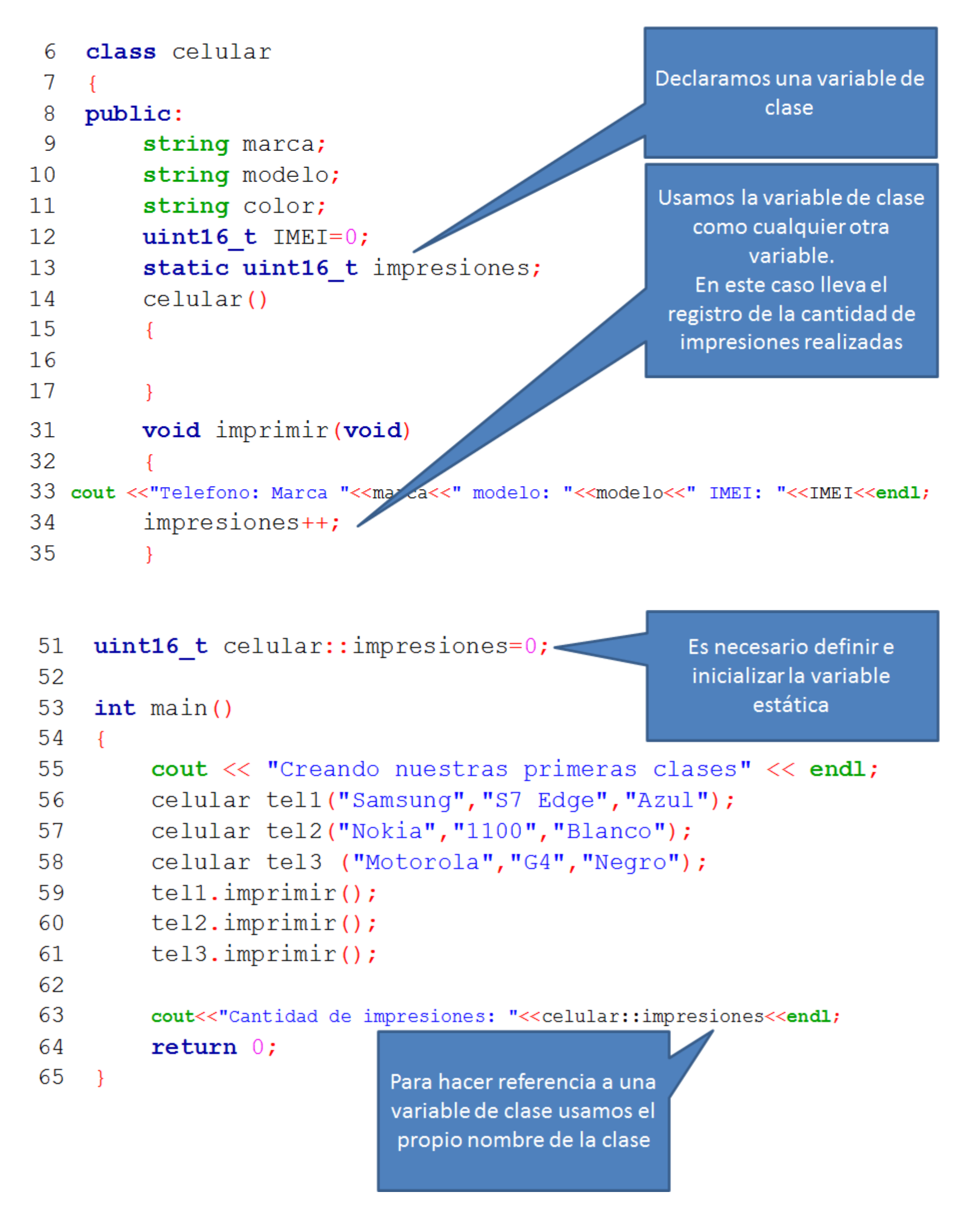


Figura 10.15

La variable `impresiones` es una variable de clase y es común a todas las instancias de la clase celular. Por eso en este ejemplo `impresiones` vale 3 ya que fue incrementada en 1 en cada una de las llamadas al método de instancia `imprimir()` (ver figura 10.16).

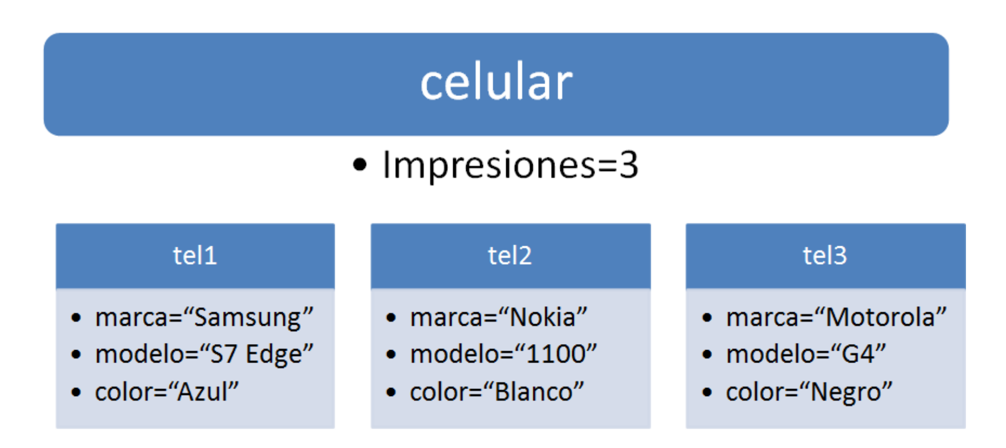


Figura 10.16

Encapsulamiento

Uno de los pilares de la POO es el encapsulamiento. Como su nombre lo indica, lo que se pretende es proteger datos y métodos dentro de las clases, escondiendo la implementación. De esta forma, como diseñador de la clase, sólo permitiremos intercambiar mensajes de manera controlada a los objetos.

En el código de la figura 10.17 se muestra la implementación de una clase producto, con el objeto de ejemplificar el concepto de encapsulamiento. Como se puede observar, la clase cuenta con los atributos `nombre` y `costo`, y un método `ImprimirEtiqueta()` que muestra en pantalla el nombre y el precio del producto luego de agregarle al costo los impuestos y la ganancia. En este ejemplo, es claro que el producto debería ocultar (encapsular) su costo para que desde fuera de la clase no se pueda acceder al mismo. Alguien, o nosotros mismos, por error podemos imprimir el costo en lugar del precio.

```
class Producto{
    public:
        string nombre;
        double costo;

        void ImprimirEtiqueta(){
            double ganancia = 30;
            double impuesto = 21;
            double precio = costo*(1+ganancia/100)*(1+impuesto/100);
            cout<<"**** "<<nombre<<" ****"<<endl;
            cout<<"*** Tan solo $"<<precio<<" ***"<<endl;
        }
};
```

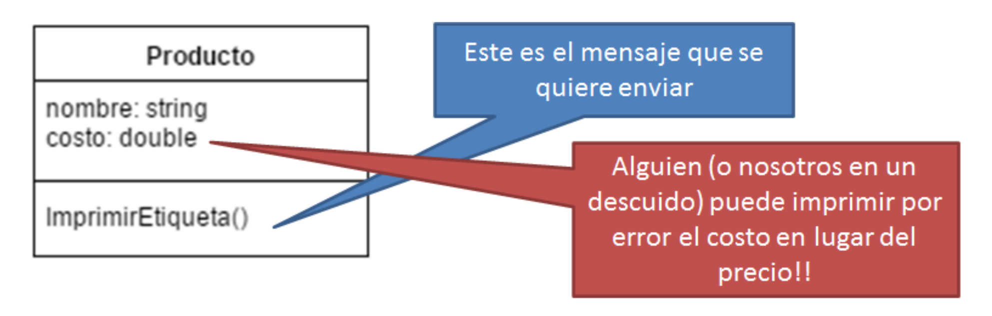


Figura 10.17

Quien diseña la clase permite enviar sólo ciertos mensajes a los objetos, es decir, mostrar una determinada interfaz para evitar que se hagan cambios “sin control” al estado del objeto y simplificar la programación al utilizar objetos que “se ven” más simples.

Para encapsular se utilizan los modificadores de acceso. Estos delimitan el comienzo de una sección dentro de la declaración de la clase. Los miembros de la clase tendrán distinto grado de acceso dependiendo dentro de qué sección son declarados:

`public`

–Puede ser accedido por cualquier clase (por defecto para los miembros de un struct)

`private`

–Sólo puede ser accedido por código dentro de la misma clase o struct (por defecto para los miembros de una clase)

`protected`

–Se comporta como private para cualquier clase y como public para clases derivadas (Se entenderá al ver la herencia).

En el código de la figura 10.18 se muestra la nueva implementación de la clase producto encapsulando el atributo costo por medio de la utilización del modificador de acceso “`private`”. Como se puede observar en la sección principal del código (main) el intento de acceso al atributo costo nos dará un error dado que no se puede acceder al atributo encapsulado desde fuera de la clase producto.

```
class Producto{
    private:
        double costo;

    public:
        string nombre;

        void ImprimirEtiqueta(){
            double ganancia = 30;
            double impuesto = 21;
            double precio = costo*(1+ganancia/100)*(1+impuesto/100);
            cout<<"**** "<<nombre<<" ****"<<endl;
            cout<<"*** Tan solo $"<<precio<<" ***"<<endl;
        }
};

    int main()
    {
        Producto p1;
        p1.nombre = "Fideos P
        p1.costo = 10;

        cout << "Etiqueta:" <<endl;
        p1.ImprimirEtiqueta();

        return 0;
    }
```

Ahora "costo" solo puede ser accedido dentro de la clase

El campo "costo", declarado como **private** dentro de la clase Producto, no puede ser accedido

```
Message
=== Build: Debug in ejemplos_clase_POO_II (
In function 'int main()':
error: 'double Producto::costo' is private
error: within this context
=== Build failed: 2 error(s), 0 warning(s)
```

Figura 10.18

En el código de la figura 10.19 se muestra una tercera implementación de la clase producto encapsulando todos sus atributos y también el método `CalcularPrecio()`.



Figura 10.19

Propiedades

Muchas veces, queremos conocer o modificar el estado de un objeto pero hemos hecho un esfuerzo por encapsular los datos y ocultar la implementación (ver figura 10.20)

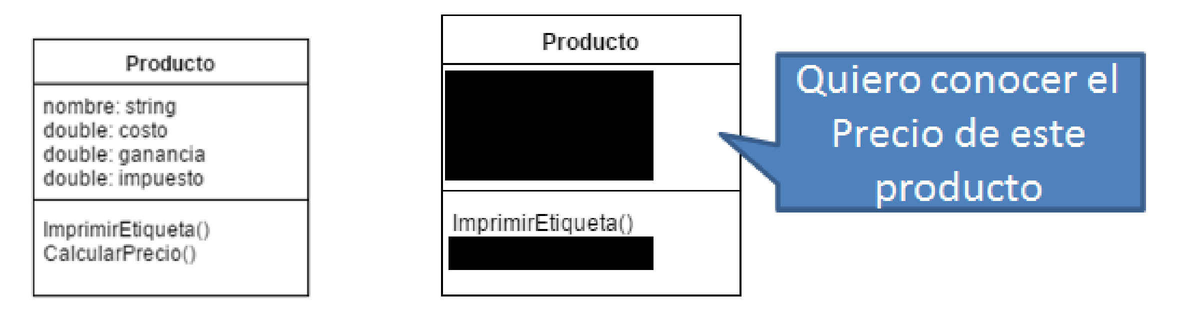


Figura 10.20

Idealmente, sólo interactuamos con un objeto por medio de su interfaz pública. La solución es implementar las “propiedades”, también conocidos como los “getter y setter” de la clase. En la figura 10.21 se presenta un ejemplo simple.

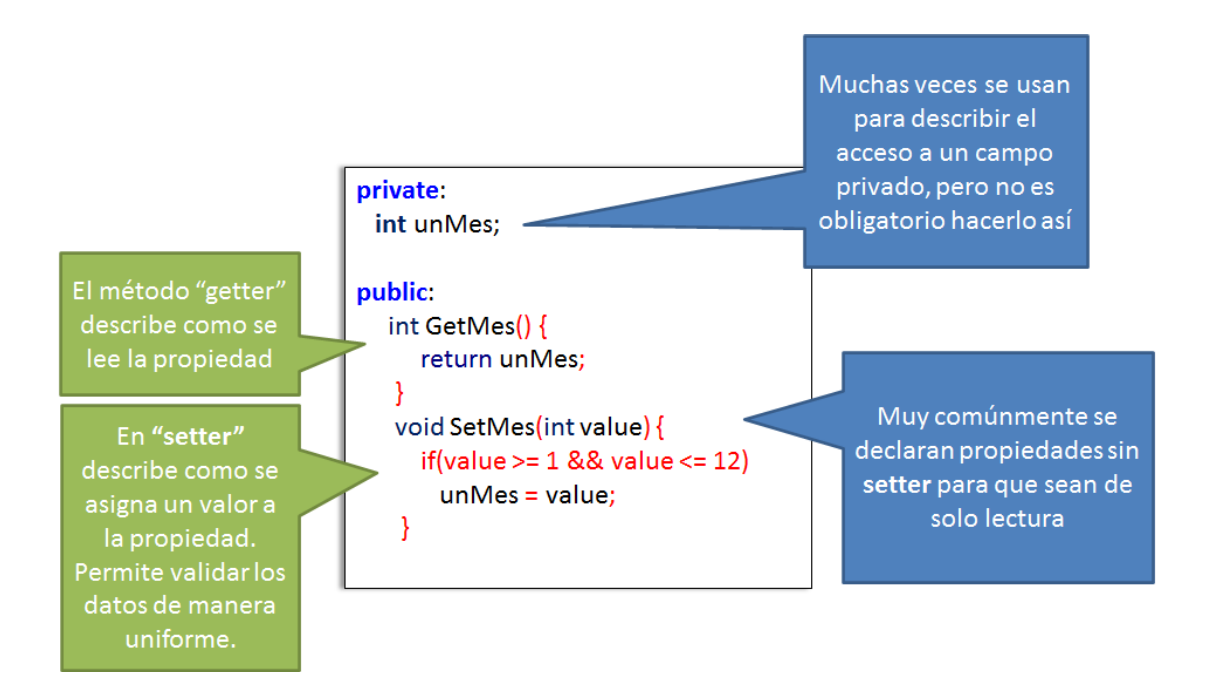


Figura 10.21

En el código que se muestra en la figura 10.22 se muestra una cuarta implementación de la clase producto encapsulando todos sus atributos e implementando propiedades para gestionar el precio del producto.

```
class Producto4{
private:
    double costo;
    string nombre;
    double costo;
    double impuesto;
public:
   Producto4(string nombre, double costo)
    {
        this->nombre = nombre;
        this->costo = costo;
         ganancia = 30;
         impuesto= 21;
    }
  void ImprimirEtiqueta(){
     cout<<"**** "<<nombre<<" ****"<<endl;
     cout<<"*** Tan solo $"<<ObtenerPrecio()<<" ***"<<e
  }

  double ObtenerPrecio(){
        return costo*(1+ganancia/100)*(1+impuesto/100);
     }
  void ModificarPrecio(double value){
        if(value>0)
            ganancia=(value/(1+impuesto/100)/costo-1)*100;
    }
};
```

Producto
ImprimirEtiqueta() double ObtenerPrecio() ModificarPrecio(double)

```
Producto4 p("Yerba La Engañosa",23.1);
p.ImprimirEtiqueta();
cout << "Acceso a la propiedad precio: ";
cout << p.ObtenerPrecio() <<endl;
p.ModificarPrecio(34);
p.ImprimirEtiqueta();
```

```
**** Yerba La Engañosa ****
*** Tan solo $36,3363 ***
Acceso a la propiedad Precio: 36,3363
**** Yerba La Engañosa ****
*** Tan solo $34 ***
```

Figura 10.22

Composición

Los lenguajes orientados a objetos tratan de simplificar la reutilización de código. La composición de clases es una herramienta para lograrlo. Con el conocimiento incorporado en lo que va del curso, se puede introducir la composición como el equivalente de estructuras anidadas pero trabajando con clases. Lo más directo es utilizar una clase anidada dentro de otra. Esto es la composición de clases. Esto no debería sorprender al lector, pues venimos usando composición al declarar un string como campo de una clase desde el inicio del capítulo.

Para ejemplificar el concepto, volvamos al primer ejemplo de la clase Producto (figura 10.23) al cual quiero modificar o agregar comportamiento. No solo quiero la clase `Producto`, sino también una clase `ProductoFresco`, que tenga fecha de vencimiento

```
class Producto{
public:
   string nombre;
   double costo;

   void ImprimirEtiqueta(){
     double ganancia = 30;
     double impuesto = 21;
     double precio =
costo*(1+ganancia/100)*(1+impuesto/100);
     cout<<"**** "<<nombre<<" ****"<<endl;
     cout<<"*** Tan solo $"<<precio<<"***"<<endl;
   }
};
```

Figura 10.23

En el código que se muestra en la figura 10.24 se implementan las clases `ProductoFresco` y `FechaVencimiento`, donde se puede observar la composición de clases. Podemos visualizar la composición observando que el producto fresco tiene una fecha de vencimiento.

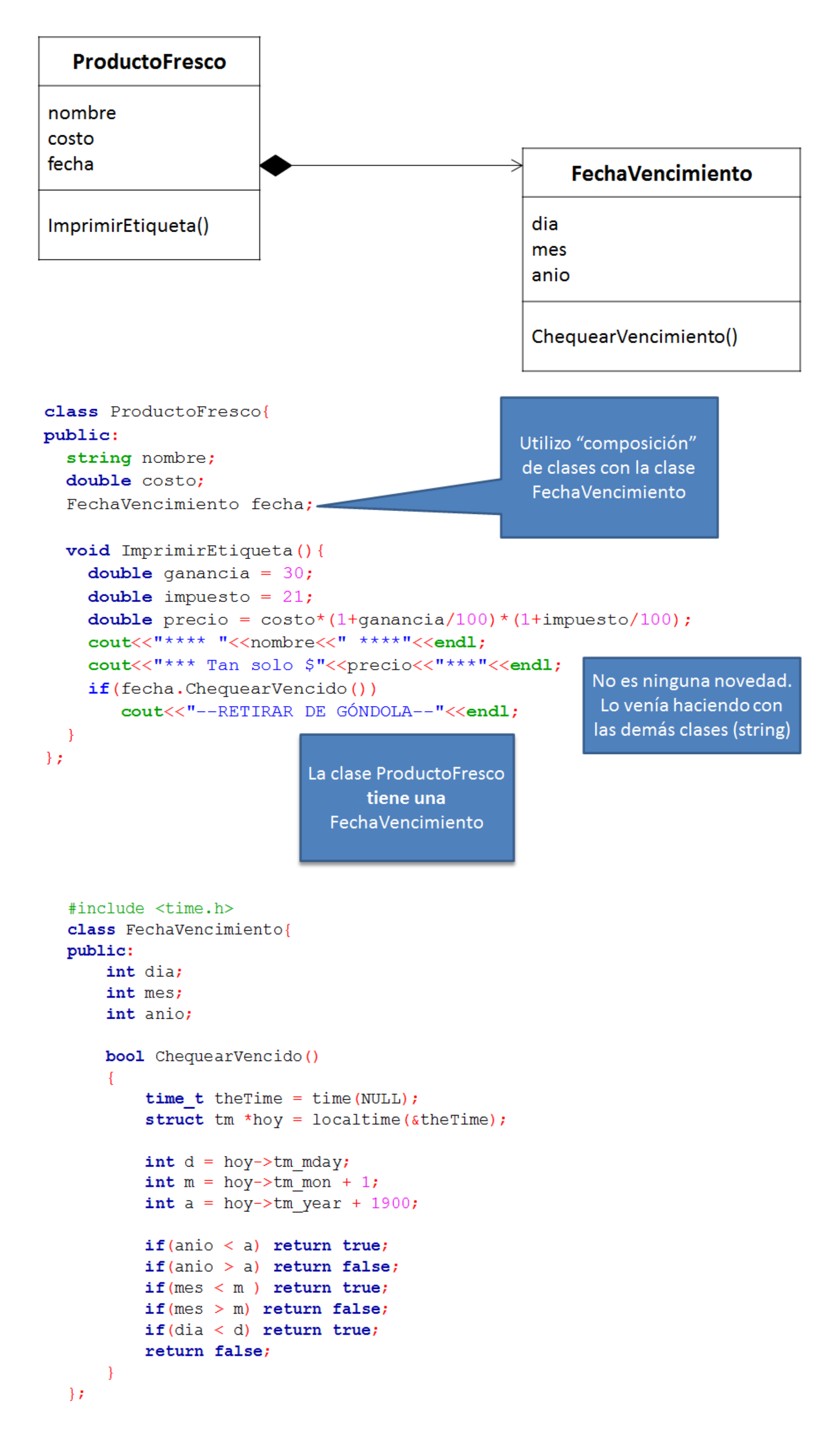


Figura 10.24

Analizando el código presentado en el ejemplo previo donde se “copia y pega” el código de `Producto` dentro de `ProductoFresco`, podemos concluir que estas dos clases están muy relacionadas, pero ambas son entidades separadas, y las tengo que mantener por separado, y usar por separado. Esto es poco eficiente y puede conducir a errores.

La solución es “componer” Producto Fresco con Producto. De esta forma reutilizamos el código evitando “copiar y pegar”. Los cambios que hagamos de ahora en más en Producto afectarán al Producto Fresco (figura 10.25).

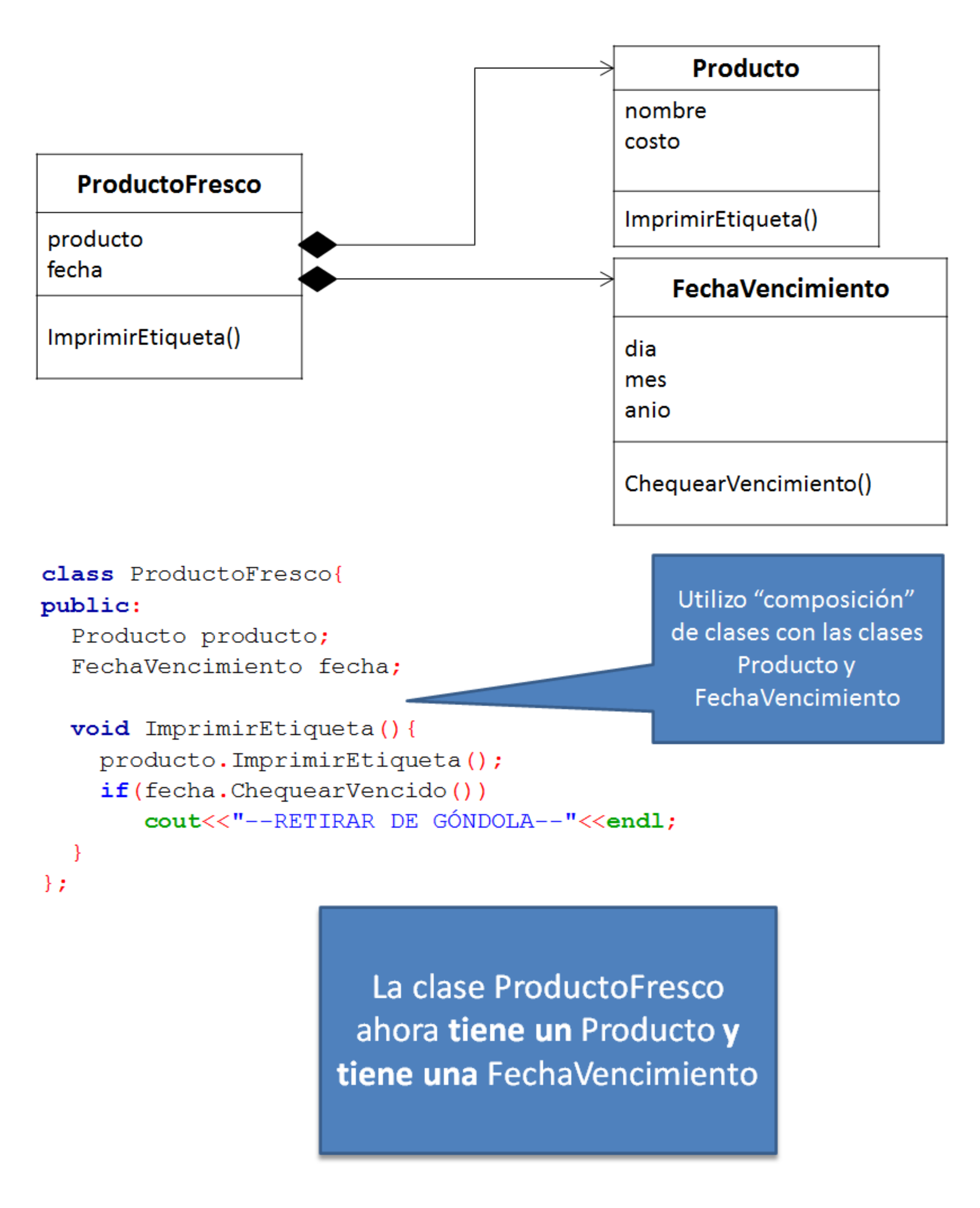


Figura 10.25

Por lo pronto, con la relación de composición utilizada, decimos que un `ProductoFresco` “tiene un” `Producto`, lo cual es raro. Para decir que ProductoFresco “es un” Producto debemos aplicar una relación de herencia entre ambas clases. A continuación se presenta el concepto de “Herencia de clases”.

Herencia

La herencia permite crear nuevas clases a partir de otras ya existentes. A partir de una clase base se pueden crear clases derivadas. Las clases derivadas tienen todos los atributos y comportamiento de su clase base. Además, pueden extenderlos y/o modificarlos.

El código para crear una clase derivada es sencillo:

```
class <Nombre clase derivada> : <acceso><nombre clase base> {
  < cuerpo >
  }
```

En la figura 10.26 se ejemplifica la implementación de una clase base simple con un único atributo y un método, y una clase derivada de la misma.

```
class ClaseBase{
public:
   int unCampo;
   void UnMetodo() { ; }
}

class ClaseDerivada : public ClaseBase {

}
```

```
ClaseBase c1;

c1.unCampo = 1;
c1.UnMetodo();

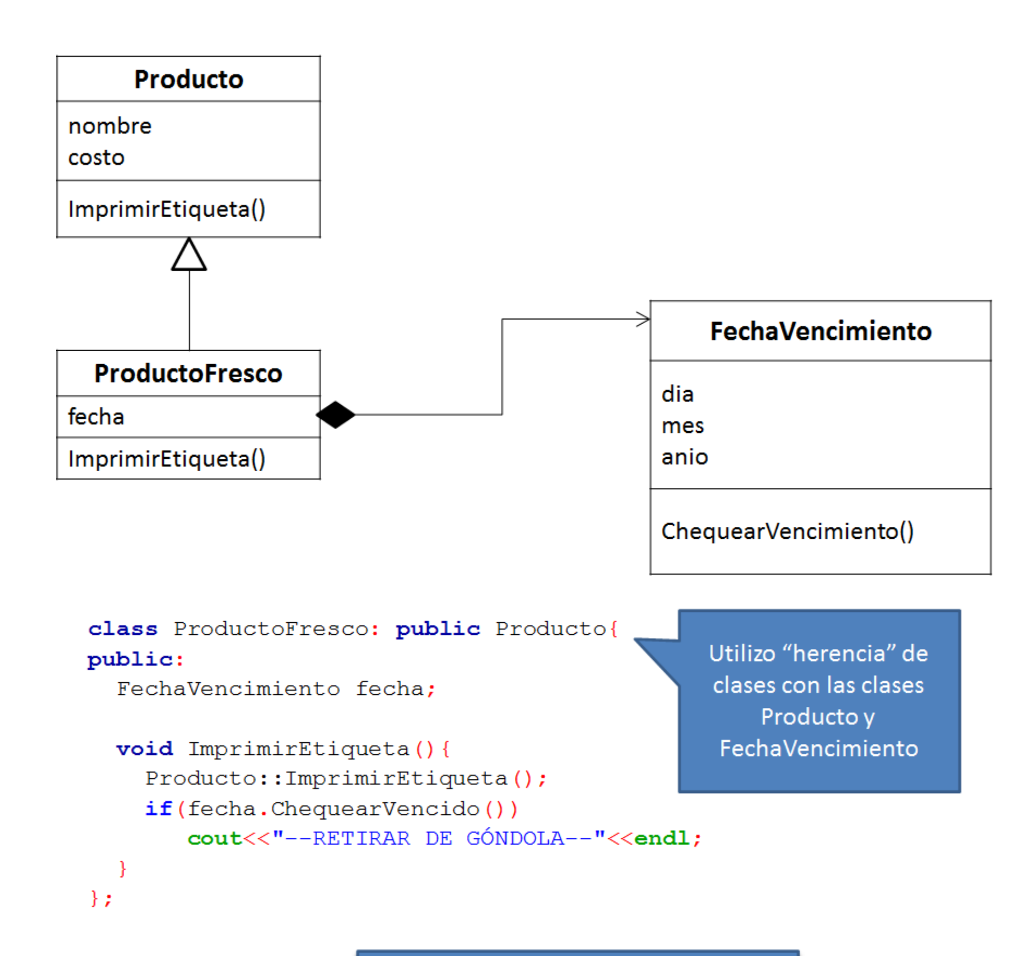
ClaseDerivada c2;
c2.unCampo  = 2;
c2.UnMetodo();
```

ClaseBase
unCampo
UnMetodo()

ClaseDerivada

Figura 10.26

Con el objeto de ejemplificar en el uso de herencia, y continuando con el ejemplo guía de la clase `Producto`, en la figura 10.27 se puede observar la implementación y uso de la clase `Producto` como clase base y la clase `ProductoFresco` como clase derivada.




```
class ProductoFresco: public Producto{
public:
  FechaVencimiento fecha;

  void ImprimirEtiqueta(){
    Producto::ImprimirEtiqueta();
    if(fecha.ChequearVencido())
       cout<<"--RETIRAR DE GÓNDOLA--"<<endl;
  }
};
```

La clase ProductoFresco ahora **es un** Producto

```
Producto p1;
p1.nombre="Yerba la engañosa";
p1.costo=23.1;

ProductoFresco p2;
p2.nombre = "Queso blanco viejo";
p2.costo = 100;
p2.fecha.dia = 10;
p2.fecha.mes=1;
p2.fecha.anio=2020;

ProductoFresco p3;
p3.nombre = "Queso blanco nuevo";
p3.costo = 160;
p3.fecha.dia = 10;
p3.fecha.mes=9;
p3.fecha.anio=2020;

p1.ImprimirEtiqueta();
cout <<endl;
p2.ImprimirEtiqueta();
cout <<endl;
p3.ImprimirEtiqueta();
```

```
**** Yerba la engañosa ****
*** Tan solo $36.3363 ***

**** Queso blanco viejo ****
*** Tan solo $157.3 ***
--RETIRAR DE GÓNDOLA--

**** Queso blanco nuevo ****
*** Tan solo $251.68 ***
```

Figura 10.27

Ahora se puede entender el especificador de acceso `Protected`:

Se comporta como `private` para cualquier clase y como `public` para las clases derivadas.

En la figura 10.28 se muestra el UML de varias clases derivadas a modo de ejemplo.

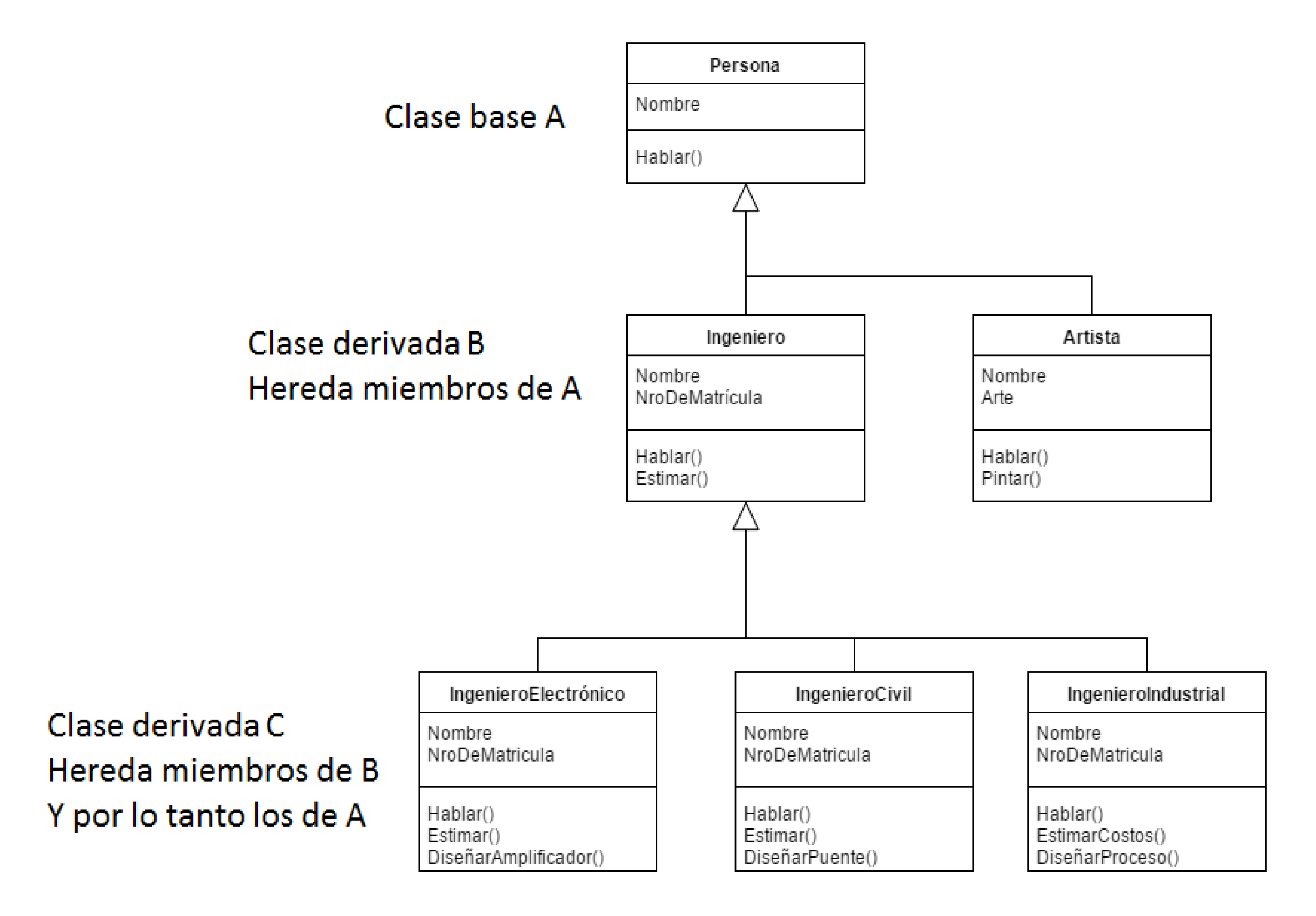


Figura 10.28

Composición y constructores

En el código que se muestra en la figura 10.29 se puede observar la forma en la cual se implementan y utilizan los constructores cuando se componen clases.

```
class ClaseBase{
private: int N;
public:
    ClaseBase(int n){
        N=n;
        cout<<"Ini. objeto tipo ClaseBase con valor "<<n<<endl;
    }

    void ImprimirN(){cout<<"N vale "<<N<<endl;}
};
```

```
class ClaseCompuesta{
private:
    char C;
public:
    ClaseBase objetoBase;
    ClaseCompuesta(int i, char c):objetoBase(i){
        C=c;
        cout<<"Ini. objeto tipo ClaseCompuesta con valores "<<i<<" "<<c<<endl;
    }
};
```

```
ClaseBase cb1(32);
cb1.ImprimirN();
ClaseDerivada cd1(21,'r');
cd1.objetoBase.ImprimirN();
```

```
Ini. objeto tipo ClaseBase con valor 32
N vale 32
Ini. objeto tipo ClaseBase con valor 21
Ini. objeto tipo ClaseCompuesta con valores 21 r
N vale 21
```

Figura 10.29

Herencia y constructores

En el código que se muestra en la figura 10.30 se puede observar la forma en la cual se implementan y utilizan los constructores cuando se heredan clases.

```
class ClaseBase{
private: int N;
public:
    ClaseBase(int n){
        N=n;
        cout<<"Ini. objeto tipo ClaseBase con valor "<<n<<endl;
    }

    void ImprimirN(){cout<<"N vale "<<N<<endl;}
};
```

```
class ClaseDerivada:public ClaseBase{
 private: char C;
 public:
    ClaseDerivada(int i, char c):ClaseBase(i){
        C=c;
        cout<<"Ini. objeto tipo ClaseDerivada con valores "<<i<<" "<<c<<endl;
    }
};
```

```
ClaseBase cb1(32);
cb1.ImprimirN();
ClaseDerivada cd1(21,'r');
cd1.ImprimirN();
```

```
Ini. objeto tipo ClaseBase con valor 32
N vale 32
Ini. objeto tipo ClaseBase con valor 21
Ini. objeto tipo ClaseDerivada con valores 21 r
N vale 21
```

Figura 10.30

Objetivo del Ejemplo integrador

Habiendo realizado una introducción a la POO y con el objetivo que el alumno afiance el modelado de datos usando objetos se presenta un ejemplo integrador.

Se pretende en primera instancia, que el alumno resuelva la consigna usando el paradigma procedural, modelando con estructuras compuestas de datos y modularizando con interfaces bien definidas. En segunda instancia, se pretende que se resuelva la misma consigna, pero modelando con POO. Por último, luego de haber realizado ambas soluciones el alumno tendrá una visión mucho más clara de ambas formas de modelar y sus diferencias.

Requerimientos del Ejemplo integrador

El ejemplo integrador pretende implementar el software embebido de un cajero automático típico. Se deberá desarrollar mediante una aplicación de consola con una interfaz de usuario bien definida. El cajero permitirá operaciones de consulta de saldo, extracciones, depósitos y transferencias. El cajero automático deberá gestionar cuentas. De las cuentas nos interesa:

- El número de cuenta.
- La contraseña de acceso.
- El saldo.
- El cliente titular de la misma.

De los clientes nos interesa conocer:

- Nombre
- Apellido
- Documento de identidad.

El cajero mantendrá su propia base de datos por medio del uso de archivos binarios.

Menú de inicio:

Cuando el sistema arranca deberá dar la bienvenida al usuario, permitiéndole acceder su número de cuenta (Figura 10.1).

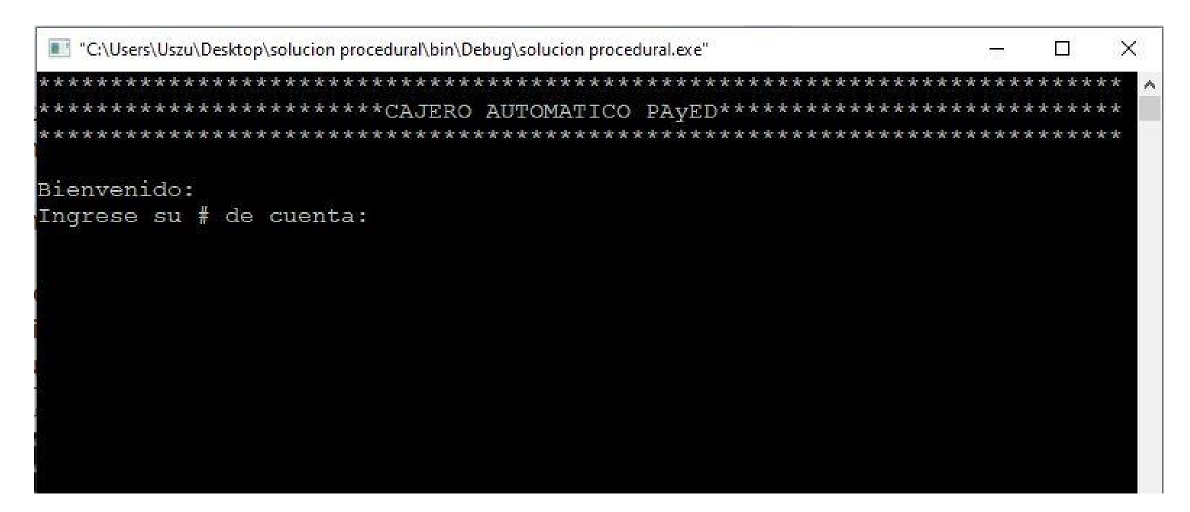


Figura 10.1. *Menú de acceso.*

En caso de que el usuario ingrese un número de cuenta válido, el sistema le dará la posibilidad de ingresar su contraseña (Figura 10.2). Caso contrario dará indicación del error y no avanzará hasta el ingreso de un número de cuenta válido.

```
"C:\Users\Uszu\Desktop\solucion procedural\bin\Debug\solucion procedural.exe"
**********************************************************************
***********************CAJERO AUTOMATICO PAyED***************************
**********************************************************************

Bienvenido:
Ingrese su # de cuenta:1000
Ingrese su contraseña:
```

Figura 10.2. *Ingreso de contraseña.*

En caso de que se ingresen el usuario y contraseña correctos, el sistema avanzará al menú principal dando la bienvenida al titular de la cuenta y presentando las opciones (Figura 10.3)

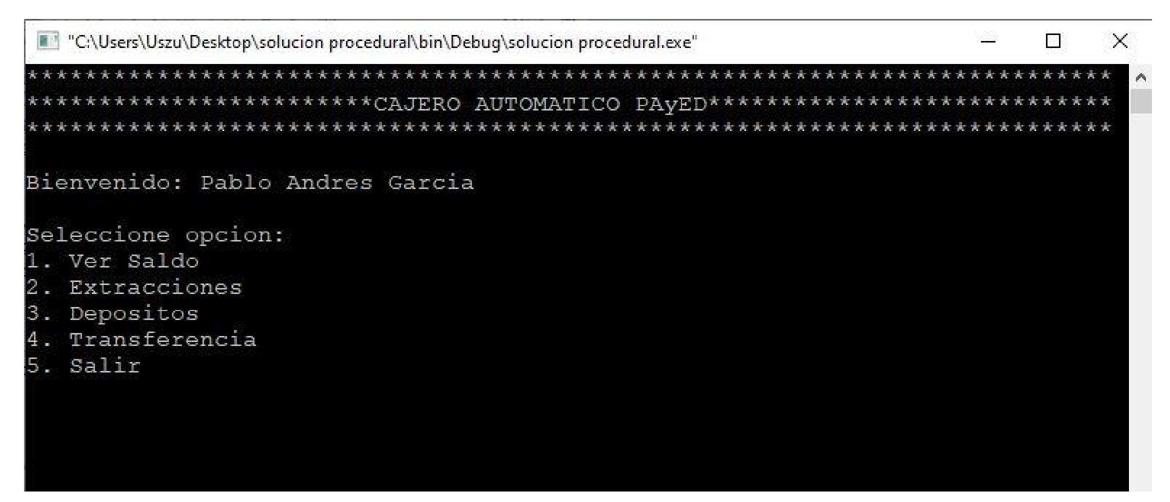


Figura 10.3. *Menú principal.*

opción 1: Ver saldo

En caso de que el usuario seleccione la opción 1, accederá a ver el saldo de su cuenta (Figura 10.4).

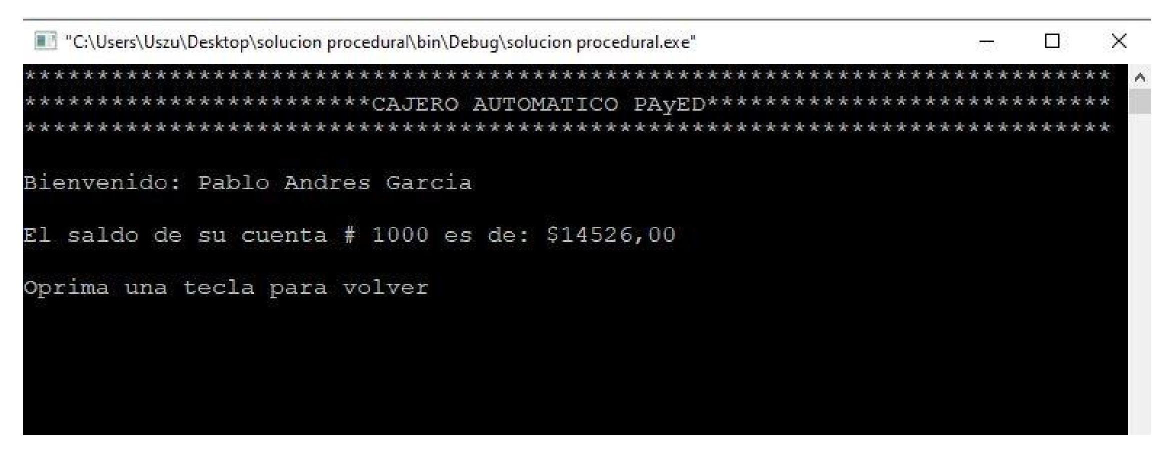


Figura 10.4. *Menú ver saldo.*

opción 2: Extracciones

En caso de que el usuario seleccione la opción 2, el sistema le dará acceso al menú de extracciones (Figura 10.5).

```
"C:\Users\Uszu\Desktop\solucion procedural\bin\Debug\solucion procedural.exe"
****************************************************************************
***********************CAJERO AUTOMATICO PAyED*****************************
****************************************************************************

Bienvenido: Pablo Andres Garcia

Seleccione importe a extraer:
1. $100
2. $200
3. $500
4. $1000
5. Otro Importe
```

Figura 10.5. *Menú extracciones.*

El usuario podrá seleccionar una de 5 opciones. El sistema deberá validar que el cliente disponga de fondos suficientes antes de realizar la extracción, y realizarla (Figura 10.6) o bien notificarlo en caso de no poder realizarla (Figura 10.7).

```
"C:\Users\Uszu\Desktop\solucion procedural\bin\Debug\solucion procedural.exe"
****************************************************************************
***********************CAJERO AUTOMATICO PAyED*****************************
****************************************************************************

Bienvenido: Pablo Andres Garcia

Seleccione importe a extraer:
1. $100
2. $200
3. $500
4. $1000
5. Otro Importe
4
Extraccion correcta, retire su dinero!
El saldo de su cuenta # 1000 es de: $13526,00

Oprima una tecla para volver
```

Figura 10.6. *Extracción correcta.*

```
"C:\Users\Uszu\Desktop\solucion procedural\bin\Debug\solucion procedural.exe"
****************************************************************************
***********************CAJERO AUTOMATICO PAyED*****************************
****************************************************************************

Bienvenido: Pablo Andres Garcia

Seleccione importe a extraer:
1. $100
2. $200
3. $500
4. $1000
5. Otro Importe
5
Ingrese importe:20000
Saldo insuficiente!!!
Oprima una tecla para volver
```

Figura 10.7. *Extracción incorrecta, saldo insuficiente.*

opción 3: Depósitos

En caso de que el usuario seleccione la opción 3 del menú principal, accede al menú de depósito (Figura 10.8). Dentro de este menú deberá ingresar el importe a depositar, retirar el sobre e incluir dentro del mismo el dinero y el comprobante impreso por el cajero. En caso de éxito deberá notificar al usuario mostrando el nuevo saldo de su cuenta.

```
"C:\Users\Uszu\Desktop\solucion procedural\bin\Debug\solucion procedural.exe"
****************************************************************************
***********************CAJERO AUTOMATICO PAyED*****************************
****************************************************************************

Bienvenido: Pablo Andres Garcia

Ingrese importe a depositar:
2000
Retire el sobre, ingrese el dinero junto con el comprobante en el sobre
Ingrese el sobre y oprima Enter para terminar

Deposito correcto!!
El saldo de su cuenta # 1000 es de: $15526,00

Oprima una tecla para volver
```

Figura 10.8. *Menú Depósitos.*

opción 4: Transferencias

En caso de que el usuario seleccione la opción 4 del menú principal, accede al menú transferencias que le permitirá transferir fondos desde su cuenta a otra cuenta existente en la base de

datos del cajero (Figura 10.9). En principio deberá ingresar un número de cuenta existente y el sistema mostrará el titular de la cuenta antes de avanzar.

```
"C:\Users\Uszu\Desktop\solucion procedural\bin\Debug\solucion procedural.exe"
****************************************************************************
***********************CAJERO AUTOMATICO PAyED*******************************
****************************************************************************

Bienvenido: Pablo Andres Garcia

Seleccione opcion:
1. Ver Saldo
2. Extracciones
3. Depositos
4. Transferencia
5. Salir
4
Ingrese el numero de cuenta a la que desar transferir:1001
El titular de la cuenta es: Guerrero Federico.
Continuar? (S/N):_
```

Figura 10.9. *Menú Transferencias.*

En caso de que el usuario valide el titular de la cuenta, el sistema avanzará al siguiente menú (FIgura 10.10).

```
"C:\Users\Uszu\Desktop\solucion procedural\bin\Debug\solucion procedural.exe"
****************************************************************************
***********************CAJERO AUTOMATICO PAyED*******************************
****************************************************************************

Bienvenido: Pablo Andres Garcia

Ingrese importe a transferir a la cuenta # 1001 del cliente Guerrero Federico
:
2500.5
Transferencia correcta!!
El saldo de su cuenta # 1000 es de: $13026,00

Oprima una tecla para volver
```

Figura 10.10. *Sub- Menú Transferencias.*

El sistema permitirá el ingreso del importe a transferir, y en caso de que el usuario disponga de saldo suficiente realizará la transferencia actualizando ambos saldos y mostrando el saldo final de la cuenta origen.

opción 5: Salir

En caso de que el usuario seleccione la opción 5, el sistema cerrará la sesión del usuario y volverá a mostrar el menú de acceso de la Figura 10.1 permitiendo el acceso a cualquier otro usuario.

Soluciones del Ejemplo integrador

Como ya se mencionó, se pretende usar el ejemplo para que el alumno logre integrar todos los conceptos incorporados a lo largo del curso. En este sentido se invita al lector a realizar su propia implementación de ambas soluciones. Para tranquilidad del lector que no lo logre, desde la cátedra se dedica una clase teórica con el objeto de describirlas.

Ejercicios

Introducción a la POO

En los siguientes ejercicios se creará una clase básica en C++ incrementando gradualmente la complejidad:

1) Declare una clase en C++ que represente los productos de un supermercado y contenga miembros nombre (cadena), marca (cadena), precio (real) y promoción (booleano). Cree un programa donde se instancie un objeto de la clase Producto, se le asignen valores a sus miembros, y se impriman en pantalla.

2) Cree una clase como la del problema 1, pero añada un método “ImprimirEtiqueta”. El método deberá devolver una cadena con el nombre y marca del producto y su precio de la siguiente manera: “Fideos PIPO, sólo $40 !!” y en el caso de que la variable promoción sea verdadera, debe imprimir “Fideos PIPO, antes $40, ahora $20!!!” (imprime la mitad de precio). Cree un programa donde se instancien 3 objetos de clase Producto, se les asignen valores a sus miembros, y se impriman las etiquetas de cada uno.

3) Agregue un constructor a la clase Producto del ejercicio anterior.

4) Cree un arreglo de Productos e implemente un programa donde se instancien objetos para cada elemento del arreglo utilizando sus constructores, y luego se impriman todas las etiquetas.

Implementación de un programa según el paradigma OO

En los siguientes ejercicios se implementará un programa para recibir datos de un sensor (que simularemos mediante una clase), procesarlos, e imprimir la información en pantalla. Se implementará la solución paso a paso y en el último de la serie se integrará en un programa completo.

5) Cree una clase "Sensor" con los métodos "Inicializar" y "LeerMuestra". El método LeerMuestra devolverá un valor real que representará una temperatura, y estará generado aleatoriamente. El método Inicializar simula la conexión del sensor, pero en nuestro programa cumplirá la función de inicializar la función aleatoria. Haga una prueba de la clase con un programa que la instancie e imprima un grupo de valores de temperatura

6) Cree una clase BufferProcesamiento, que tendrá la capacidad de almacenar un N valores reales (N configurado según un parámetro pasado a su constructor y almacenado como campo) y realizar operaciones sobre esos valores. Tendrá un método cargarValor que permitirá almacenar un nuevo valor hasta que el buffer se llena, un método BufferLleno que devolverá verdadero si se completaron los N valores, Vaciar, y métodos ObtenerMedia, ObtenerMáximo, y ObtenerMínimo. Cree un programa de prueba.

7) Cree una clase RegistroTemperaturas, que será capaz de almacenar una lista con una hora y temperatura media, máxima y mínima y leer y almacenar en un archivo o imprimir en pantalla la información recabada. Cree un programa de prueba.

8) Integre las clases anteriores en un programa que lea una temperatura por segundo y luego de un minuto (o sea, de leer N=60 muestras) imprima en consola la hora y minutos y los valores medios, mínimos y máximos de temperatura

Diseño de programas según el paradigma OO

Diseñe soluciones a los siguientes problemas basadas en el paradigma OO.

9) Cree un programa que implemente una calculadora con la posibilidad de hacer las operaciones básicas aritméticas, factorial, y saber si un número es primo, y que pueda guardar los cálculos realizados (no sólo el resultado, sino también las operaciones que se introdujeron) en un historial.
Separe en clases correspondientes la calculadora en sí, la interfaz de usuario, el historial, y las clases que estas a su vez requieran.

Bibliografía ampliatoria

Cairó Battistutti, Osvaldo, "Metodología de la programación", 3a. Ed. Alfaomega, 2005.

Deitel, H. M. and Deitel P. J., "Cómo programar en C", 2da. Ed., Pearson Educación, 2007.

Englander, I (2003). The architecture of computer hardware and systems software: an information technology approach. Wiley, 2003.

Greg Perry, "C con Ejemplos", Prentice Hall, 2000.

Gottfried, Byron "Programación en C", 2da. Ed McGraw Hill, 1997.

Schildt, Herbert "C, Manual de referencia", Mc Graw Hill 1996.

Kernighan Brian W. y Ritchie Dennis M. "El lenguaje de programación C", 2da. Ed. Prentice Hall 1991.

Louden, Programming Languages: Principles and Practice, 3ed. 2011.

Sebesta, Concepts of Programming Languages,11ed, Pearson 2015.

Silberschatz, A., Galvin P., y Gagne G. (2009). Operating system concepts with Java. Wiley Publishing, 2009.

Tanenbaum, A. S., y Bos H. (2015). Modern operating systems. Pearson, 2015.

Wirth, Niklaus, "Introducción a la Programación Sistemática", editorial El Ateneo, 1982.

Los Autores

Coordinadores

García, Pablo A.

Nació en Azul, Provincia de Buenos Aires, en 1976. Recibió los títulos de Ingeniero en electrónica, Magister en ingeniería y Doctor en ingeniería en la Universidad Nacional de La Plata (UNLP), en 2002, 2008 y 2019 respectivamente. Actualmente es Profesor Titular de la cátedra Programación-E1201 y ayudante diplomado de la cátedra Proyecto Final-E1227 en el departamento de Electrotecnia de la Facultad de Ingeniería en la UNLP. Es coordinador y docente del curso de posgrado "Fundamentos de la Programación Orientada a Objetos" y forma parte del grupo docente del curso de posgrado "Procesamiento analógico de señales" (FI, UNLP). Integra el grupo de Instrumentación biomédica, Industrial y Científica (GIBIC) dependiente del Instituto de Investigaciones en Electrónica, Control y Procesamiento de Señales (LEICI) de la UNLP y el CONICET.

Haberman, Marcelo A.

Nació en La Plata, Provincia de Buenos Aires, en 1984. Recibió los títulos de Ingeniero en electrónica y Doctor en ingeniería en la Facultad de Ingeniería de la Universidad Nacional de La Plata (FI-UNLP), en 2008 y 2016 respectivamente. Actualmente es Profesor Adjunto Suplente de la cátedra de Programación-E1201 (FI-UNLP), donde trabaja desde 2006, siendo previamente Ayudante y Jefe de Trabajos Prácticos. Es parte del equipo docente del curso de posgrado "Fundamentos de la Programación Orientada a Objetos" y Ayudante Diplomado en Materiales y Componentes Electrotécnicos-E0207. Es Investigador Asistente del Consejo Nacional de Investigaciones Científicas y Técnicas (CONICET) desde 2016, con lugar de trabajo en el grupo de Instrumentación biomédica, Industrial y Científica (GIBIC) dependiente del Instituto de Investigaciones en Electrónica, Control y Procesamiento de Señales (LEICI).

Guerrero, Federico N.

Nació en Comodoro Rivadavia, Provincia de Chubut, en 1986. Recibió los títulos de Ingeniero en electrónica y Doctor en ingeniería en la Facultad de Ingeniería de la Universidad Nacional de La Plata (UNLP), en 2011 y 2017 respectivamente. Actualmente es Investigador Asistente del Consejo Nacional de Investigaciones Científicas y Técnicas (CONICET) con lugar de trabajo en el Instituto de Investigaciones en Electrónica, Control y Procesamiento de Señales LEICI (UNLP-

CONICET-CIC), donde ejecuta su plan de investigación sobre instrumentación para biopotenciales. El Dr. Guerrero es Jefe de Trabajos Prácticos Suplente en la cátedra de Programación (Dpto. de Electrotecnia, FI, UNLP), donde ha sido Colaborador, Ayudante Interino y Ayudante Ordinario sucesivamente desde el año 2012. Es parte del equipo docente del curso de posgrado “Fundamentos de la Programación Orientada a Objetos” (FI, UNLP).

Autores

Mendez, Leandro

Nació en La Plata, Provincia de Buenos Aires, 1978. Recibió el título de Ingeniero en Electrónica, en la Universidad Nacional de la Plata (UNLP), en 2006. Actualmente es Ayudante diplomado en la cátedra de Programación E-1201, en el Departamento de Electrotecnia de la Facultad de Ingeniería en la UNLP.

Moyano, Raúl Alejandro

Ingeniero Electrónico por la UNLP y Miembro del Plantel Docente de la Cátedra de PAyED (UNLP) y de Control Moderno (UNLP) y de Control y Servomecanismos B (UNLP).

Rosso, Juan M.

Nació en la ciudad de Rosario, Provincia de Santa Fe, en 1977. Recibió en 2002 el título de Ingeniero en Electrónica en la Facultad de Ingeniería de la Universidad Nacional de La Plata (UNLP). Desde el año 2006 a la actualidad participa de la cátedra Programación-E1201 de la Facultad de Ingeniería de la UNLP como ayudante Diplomado. Además, dirige el Departamento de Diseño y Programación de equipos electrónicos en una empresa dedicada a la fabricación de instrumentos de medición y control para la industria.

Libros de Cátedra

Programación E1201 : curso de grado / Pablo Andrés García ... [et al.] ; coordinación general de Pablo Andrés García ; Marcelo Alejandro Haberman ; Federico Nicolás Guerrero ; prólogo de Graciela Toccaceli. - 1a ed. - La Plata : Universidad Nacional de La Plata ; EDULP, 2021.
Libro digital, PDF

Archivo Digital: descarga
ISBN 978-950-34-2034-8

1. Lenguajes de Programación. 2. Algoritmo. 3. Ingeniería. I. García, Pablo Andrés, coord. II. Haberman, Marcelo Alejandro, coord. III. Guerrero, Federico Nicolás, coord. IV. Toccaceli, Graciela, prolog.
CDD 005.1301

Diseño de tapa: Dirección de Comunicación Visual de la UNLP

Universidad Nacional de La Plata – Editorial de la Universidad de La Plata
48 N.º 551-599 / La Plata B1900AMX / Buenos Aires, Argentina
+54 221 644 7150
edulp.editorial@gmail.com
www.editorial.unlp.edu.ar

Edulp integra la Red de Editoriales Universitarias Nacionales (REUN)

Primera edición, 2021
ISBN 978-950-34-2034-8





